Understanding and Evolving the ML Module System

Derek Dreyer

May 2005
CMU-CS-05-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Robert Harper (co-chair)
Karl Crary (co-chair)
Peter Lee
David MacQueen (University of Chicago)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored in part by the National Science Foundation under grant CCR-0121633 and EIA-9706572, and the US Air Force under grant F19628-95-C-0050 and a generous fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: ML, module systems, type systems, functors, abstract data types, lambda calculus, recursive modules, singleton kinds

Abstract

The ML module system stands as a high-water mark of programming language support for data abstraction. Nevertheless, it is not in a fully evolved state. One prominent weakness is that module interdependencies in ML are restricted to be acyclic, which means that mutually recursive functions and data types must be written in the same module even if they belong conceptually in different modules. Existing efforts to remedy this limitation either involve drastic changes to the notion of what a module is, or fail to allow mutually recursive modules to hide type information from one another. Another issue is that there are several dialects of ML, and the module systems of these dialects differ in subtle yet semantically significant ways that have been difficult to account for in any rigorous way. It is important to come to a clear assessment of the existing design space and consolidate what is meant by "the ML module system" before embarking on such a major extension as recursive modules.

In this dissertation I contribute to the understanding and evolution of the ML module system by: (1) developing a unifying account of the ML module system in which existing variants may be understood as subsystems that pick and choose different features, (2) exploring how to extend ML with recursive modules in a way that does not inhibit data abstraction, and (3) incorporating the understanding gained from (1) and (2) into the design of a new, evolved dialect of ML. I formalize the language of part (3) using the framework of Harper and Stone, in which the meanings of "external" ML programs are interpreted by translation into an "internal" type system.

In my exploration of the recursive module problem, I also propose a type system for statically detecting whether or not recursive module definitions are "safe"—that is, whether they can be evaluated without referring to one another prematurely—thus enabling more efficient compilation of recursive modules. Future work remains, however, with regard to type inference and type system complexity, before my proposal can be feasibly incorporated into ML.

For Constance and Benard Dreyer,
the most loving and supportive parents in the world

Acknowledgments

First and foremost, I would like to thank my advisors, Bob Harper and Karl Crary. Many of the ideas in this thesis were developed together with them, and my character as a researcher has been influenced to a large degree by their rigorous approach to programming language research and their profound sense of aesthetics. I would also like to thank Peter Lee for his warm guidance as my former advisor and his continual encouragement of all my endeavors, and Dave MacQueen for many interesting discussions and for his extremely careful and thorough perusal of this dissertation.

There are many other friends and colleagues whom I would like to thank for making my experience at CMU such a memorable (and long!) journey. To name a few: Melissa and Umut Acar, Mihai and Raluca Budiu, Sharon Burks, Franklin Chen, Andrés Cladera, Catherine Copetas, Kathy Copic, Nathaniel Daw, Mark Fuhs, Anna Goldenberg, Beth and Jeff Helzner, Heather Hendrickson, Rose Hoberman, Yan Karklin, Cathy Kelley, Adam Klivans, Sue Lee, Adriana Moscatelli, Mike Murphy, Tom Murphy VII, Aleks and Emi Nanevski, Leaf Petersen, Martha Petersen, Frank Pfenning, Chris Richards, Chuck Rosenberg, Dan Spoonhower, Chris Stone, Dave Swasey, Desney Tan, Joe Vanderwaart, Dave Walker, Kevin Watkins.

I would like to give special thanks to Aleks Nanevski. I am very lucky to have had such an intellectually stimulating (if occasionally infuriating) officemate for seven whole years. I will always remember our spirited discussions about life, love, politics and programming languages with great fondness.

I would also like to thank Kevin Watkins for initiating, and Dan Spoonhower for revamping, the ConCert reading group. One of the strengths of CMU is its strong graduate student community, and it was wonderful to be able to hash out the details of research papers with such an informed and motivated group of colleagues.

Finally, I would like to thank Rose Amanda Hoberman for all the love, companionship and delicious food she has given me over the past four and a half years, and my parents, Constance and Benard, for all the love and encouragement they have given me throughout my life.

Contents

List of Figures

Introduction

Nearly all programming languages that are intended for the implementation of real-world applications provide some facility for structuring programs as a network of smaller modules. While structuring a program in this fashion typically incurs some initial development overhead, it also reaps several huge rewards. First, it allows multiple programmers to work on different modules of the same program simultaneously. Second, it delineates the high-level structure of the program, which in turn makes the program easier to understand and maintain. Third, it can make the program significantly more reliable through the use of *data abstraction*.

The idea of data abstraction is that the weaker the dependencies between program modules, the more robust the program structure will be. In particular, for the purpose of enforcing program invariants, it is useful for the implementor of a module to be able to hide information from clients of the module regarding the structure of data that it operates on. For example, the correctness of a module implementing binary search trees as red-black trees depends on the invariant that the trees it operates on have no two adjacent red nodes. The tree operations provided by the module assume this invariant about the trees they are given as input and preserve this invariant for the trees they produce as output. To ensure that the input assumptions are valid, the implementor should be able to restrict the clients of the module so that they may only construct red-black trees via the invariant-preserving operations that the module provides. Such a restriction, which constitutes a weakening of the dependency between the module and its clients, also makes the client code more reusable. One may make arbitrary changes to the implementation of the binary search tree module without precipitating changes to its clients, so long as the external functionality of the module—*i.e.,* the set of operations it provides—remains the same.

Many modern programming languages provide some form of support for data abstraction. For example, in mainstream object-oriented languages like C++ and Java, "classes" support data abstraction by allowing certain fields or methods of a class to be designated as "private" and therefore invisible to the clients of the class. However, classes also embody a host of other features of object-oriented programming, such as inheritance, subtyping and dynamic dispatch. The subtle and sometimes undesirable interplay of these features makes classes a tricky object for formal study.

In this dissertation I will be exploring an altogether different approach to modular programming, namely that of the ML module system. Unlike the class mechanism in object-oriented languages, the module mechanism in ML is focused entirely on supporting data abstraction. It ensures *implementor-side* data abstraction by allowing the implementor of a module to "seal" it behind an abstract interface, thereby hiding information about its internal data representation from its clients. Furthermore, ML's notion of an interface is very flexible, enabling one to reveal partial information about the identity of an abstract data type or to express equivalences between abstract data types exported by different modules. ML also exploits a form of *client-side* data abstraction through the "functor" mechanism. Functors, which are functions at the level of modules, allow one to develop and compile a module independently from the modules on which it depends, given only abstract interfaces for them. These dependencies can then be instantiated with multiple different modules during the execution of the program, enabling a powerful form of code reuse.[1]

Nevertheless, despite its strengths, the ML module system is not in a fully evolved state. One prominent weakness is that module interdependencies in ML are restricted to be acyclic, which means that mutually recursive functions and data types must be written in the same module even if they belong conceptually in different modules. In addition to hindering independent development of mutually recursive components, this restriction inhibits data abstraction because it prevents mutually recursive components from hiding implementation details from one another. Furthermore, although there are justifications for ML's restriction, it still seems rather unintuitive to most newcomers to the language, who are accustomed to languages like Java and C++ that do allow cyclic dependencies between program components. As a consequence, support for mutually recursive modules has been one of the most frequently requested extensions to ML.

In response, there has been much work in recent years on extending ML and other functional languages with recursive modules. Most of the current proposals suggest replacing ML modules with some alternative mechanism such as "units" or "mixins," which are subject to severe syntactic restrictions but, as a result, are easier to recursively link [20, 16]. The only concrete proposals that remain within the framework of ML modules are Russo's extension to the Moscow ML compiler [66] and Leroy's extension to the Objective Caml compiler [44]. Both of these are based to a large extent on Crary *et al.*'s foundational account of recursive modules [6]. Neither extension, however, provides full support for data abstraction between, or separate compilation of, mutually recursive modules.

Before we can consider ways of improving on the existing proposals for extending ML with recursive modules, there are two more basic questions that need to be addressed. First, what language should we be extending? There is not just one language called ML; there are several dialects of ML, the most popular being Standard ML (SML) [52] and Objective Caml (O'Caml) [41]. These implemented dialects are both the result of and inspiration for a large body of research on the theoretical underpinnings of ML and in particular its module system. However, as these formal accounts of the ML module system employ a variety of different formalisms, the relationships and tradeoffs between different designs have been difficult to understand or compare in a rigorous way. It is important that we come to a clear assessment of the existing design space and consolidate what is meant by "the ML module system" before embarking on such a major extension as recursive modules.

Second, once we decide on the basis for our extension, what is the right way to go about defining the extension? Type theory and operational semantics have proven to be an ideal setting for defining and reasoning about fundamental concepts like polymorphism, data abstraction and subtyping, with established methods for proving properties like type safety and decidability of typechecking. However, while there exist a number of type-theoretic accounts of ML-style modularity, they typically describe some idealized subset of the ML language. On the other hand, the Standard ML dialect has been given a full formal definition, and the very existence of the Definition of SML [52] has encouraged the development of independent implementations of the language while providing stability of SML code across those implementations. The flip side of that stability is that the Definition is closely tailored to the needs of SML, often to the point of seeming *ad hoc* from a more general semantic perspective. For example, it is not clear how the "semantic object" language that the Definition uses to formalize SML's static semantics corresponds to traditional type structures.

The approach I take in this dissertation follows the work of Harper and Stone [33], who give an alternative interpretation of Standard ML by translating well-formed SML programs into a type

[1] I will give concrete examples of how ML's sealing and functor mechanisms are used in Chapter 1. For a detailed comparison of the ML approach and the object-oriented approach to modularity, see MacQueen [48].

theory. The Harper-Stone approach provides one with a flexible method of evolving a full-fledged programming language. The key concepts of the language are modeled at the level of the type theory, where they can be more clearly understood. The features of the language that are not so much semantically interesting as syntactically convenient—*e.g.,* type inference, pattern matching, the `open`'ing of a module's namespace—are handled by the translation (called *elaboration*), which formalizes how these features are to be de-sugared into more basic constructs.

Another advantage of the Harper-Stone approach is that it fits neatly into the model of type-directed compilation [31, 77, 70, 68]. In traditional compilers, type information is discarded after typechecking. In type-directed compilers, the intermediate languages of the compiler are typed so as to enable optimizations that rely on type information. For instance, the TILT compiler for SML developed at CMU [77, 74, 61] maintains type information throughout compilation in order to implement intensional type analysis [31] and tag-free garbage collection [55]. Thus, regardless of how ML programs are typechecked, a type-directed ML compiler will at some point need to translate them into an internal typed language. TILT performs this translation as part of typechecking, based closely on Harper and Stone's elaboration algorithm.

In summary, the goal of this dissertation is to contribute to the evolution of the ML module system in the following ways:

Part I: To develop a unifying account of existing variants of the ML module system that can serve as a basis for future research on module systems.

Part II: To explore the problem of extending ML with recursive modules, with the goal of emending the deficiencies of existing proposals.

Part III: To formally define a language based on the work of Parts I and II within the type-theoretic framework of Harper and Stone.

The document is structured as follows:

Part I: In Chapter 1, I give an overview of the design space of ML modules, as well as a discussion of the key ideas and motivations behind a variety of existing designs. In Chapter 2, I give a high-level semantic analysis of what makes ML-style data abstraction work, leading to a unifying framework in which several variants of the ML module system can be understood as subsystems that pick and choose different features. I also describe the design of a type system, based on this unifying framework, that harmonizes and improves on the existing designs. This type system is formally defined in Chapters 3 and 4: the first chapter presents the "core" language of the type system, and the second chapter presents the "module" language. In addition to providing detailed discussion of the rationale behind various typing rules, these chapters present the key meta-theoretic properties of the core and module languages, which include type safety and decidability of typechecking.

Part II: In Chapter 5, I give an overview of the design space of existing recursive module extensions to ML and discuss the deficiencies of these proposals, which suggest two key directions for improvement. One direction for improvement involves the interaction of recursive modules and data abstraction. I describe (in Chapter 5) an intuitive semantics that allows for real data abstraction between recursive modules. Some aspects of this semantics can be captured in type theory, and in Chapter 6 I show how to extend the type theory of Chapters 3 and 4 accordingly. However, there is a significant component of my intuitive semantics for recursive modules that I do not know how to account for directly in type theory. I formalize it instead in Part III using elaboration techniques. The other direction for improvement on existing recursive module designs involves statically detecting whether a recursive module is "safe." In Chapter 7, I explore this direction in the context of the simply-typed λ-calculus, setting aside the orthogonal issues involving type components in modules. At the end of the chapter, I discuss what would be involved in scaling my proposed approach to the level of modules with type components.

Part III: In Chapters 8 and 9, I use the Harper-Stone framework as a starting point for defining a new dialect of ML. Following Harper-Stone, the language is defined in two parts. The "internal" language, defined in Chapter 8, is a type system based very closely on the work of Chapters 3, 4 and 6. The "external" programmer-level language, defined in Chapter 9 by translation into the internal language, is an evolved dialect of Standard ML that supports recursive and higher-order modules. Finally, in Chapter 10, I conclude and suggest directions for future work.

Part I

Understanding the ML Module System

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 "$\mathsf{open}\ v\ \mathsf{as}\ [\alpha, x]\ \mathsf{in}\ e$"—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.

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 $\Longleftrightarrow$ `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` $\quad (\mathrm{M}_{\mathrm{GUI}})$

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 $\mathrm{M}_{\mathrm{GUI}}$`.t` does not make any sense. One evaluation of $\mathrm{M}_{\mathrm{GUI}}$ 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 $\mathrm{M}_{\mathrm{GUI}}$`.t` is not well-defined.

So we see that there are certain module expressions like $\mathrm{M}_{\mathrm{GUI}}$ 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 $\mathrm{M}_{\mathrm{GUI}}$ (defined above) in place of M, then it would be unsound to treat `X.t` and `Y.t` as equivalent types.

[1]It 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 $\mathrm{M_{GUI}}$ 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
```
$(\mathrm{M_{VAL}})$

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
```
$(\mathrm{M_{REF}})$

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 $\mathrm{M_{REF}}$ 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 $\mathrm{M_{GUI}}$ 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
```
$(\mathrm{M_{DICT}})$

The difference in the conditional test between $\mathrm{M_{GUI}}$ and $\mathrm{M_{DICT}}$ is a significant one. Instead of querying the state of the GUI, $\mathrm{M_{DICT}}$ checks whether `n` is less than 20, which is a pure operation! Consider plugging $\mathrm{M_{DICT}}$ in for M in the scenario of Figure 2.1 (where I assume static scoping, so that both copies of $\mathrm{M_{DICT}}$ 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 $\mathrm{M_{DICT}}$ depends on the dynamic purity of `n`, *i.e.,* that `n` is instantiated by some *value*.

[2] A 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.

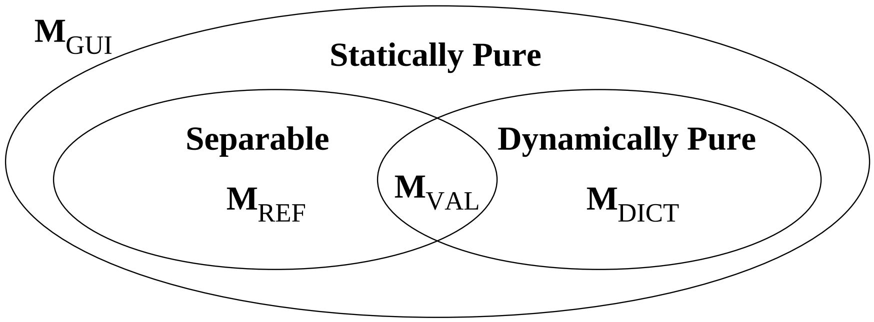

Figure 2.2: Classifications of Module Expressions

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 `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 `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 $\mathrm{M}_{\mathrm{DICT}}$ 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, $\mathrm{M}_{\mathrm{VAL}}$ and $\mathrm{M}_{\mathrm{REF}}$ are separable, while $\mathrm{M}_{\mathrm{GUI}}$ and $\mathrm{M}_{\mathrm{DICT}}$ 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 $\mathrm{M}_{\mathrm{GUI}}$ through $\mathrm{M}_{\mathrm{DICT}}$ 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

[3] One 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 `t` component relies on the evaluation of code in M, as is the case for $\mathrm{M}_{\mathrm{DICT}}$`.t`.

[4] The 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.

[5] I 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 `t`. How do we determine if $\mathrm{M}.\mathtt{t} = \mathrm{N}.\mathtt{t}$? If the signatures of M and N specify the identity of `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 `t` opaquely. In that case, the answer is that $\mathrm{M}.\mathtt{t} = \mathrm{N}.\mathtt{t}$ 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 $\mathrm{M}_{\mathrm{DICT}}$ defined above is statically (and dynamically) equivalent to itself, but only under *dynamically* equivalent instantiations of its free variable `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.

[6] One can think of the integer `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 `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.t` equal to `Y.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 M_DICT end
```

[7]In 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.

If `F` is:	And N is:	Then `F(N)` is:
separably total	separable	separable
separably total	statically pure	statically pure
dynamically total	dynamically pure	dynamically pure
statically total	dynamically pure	statically pure

If `F` is:	And N_1 and N_2 are:	Then `F(`N_1`) and F(`N_2`)` are:
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 M_{DICT} 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 M_{DICT} 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
```
$(\mathrm{M}_{\mathrm{SEAL}})$

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 $\mathrm{M}_{\mathrm{SEAL}}$ does not have any actual run-time effect, the evaluation of $\mathrm{M}_{\mathrm{SEAL}}$ will always result in a module value that defines `set` to be the same type, `int list`. Thus, $\mathrm{M}_{\mathrm{SEAL}}$ 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 $\mathrm{M}_{\mathrm{SEAL}}$ 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 $\mathrm{M}_{\mathrm{SEAL}}$, 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 $\mathrm{M}_{\mathrm{SEAL}}$ 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

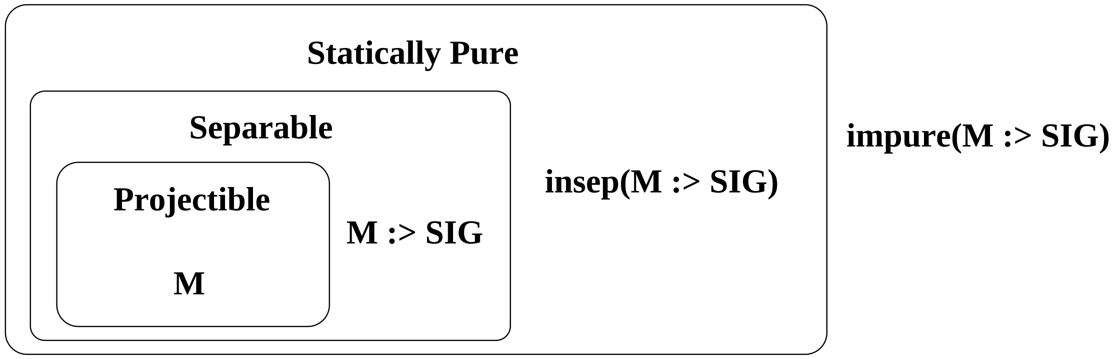

Figure 2.4: Semantic Effects of Sealing

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 `impure(M :> 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 `insep(M :> 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

[8] One 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 `OpaqueG` as total, too.

The reader may wonder: what is the point of `OpaqueG` 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 `OpaqueG`. If `G` is considered partial, *i.e.,* if we do not observe that transparent modules are pure, then the body of `OpaqueG` will be considered impure as well, and `OpaqueG` will be treated as partial. In that case, repeated applications of `OpaqueG` will *not* result in modules with equivalent type components because `OpaqueG`'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 $\mathrm{M_{DICT}}$ that are statically pure but inseparable are treated the same as modules like $\mathrm{M_{GUI}}$ 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 $\mathrm{M_{GUI}}$ and $\mathrm{M_{DICT}}$ 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.

[9] The 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" $\langle\!|\mathrm{S}|\!\rangle$, 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 $\mathrm{M}_{\mathrm{GUI}}$ and $\mathrm{M}_{\mathrm{DICT}}$ 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 $\langle\!|$`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 `unpack`ing. Since the type components of an `unpack`ed module expression may depend on the result of a core-level dynamic computation, I conservatively treat all `unpack`ed 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 `unpack`ing.

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 $\mathsf{Fst}(M)$, that computes a type constructor representing the "static part" of M. When M is a module variable `X`, $\mathsf{Fst}(M)$ is a constructor variable $\mathtt{X}^{\mathsf{c}}$, which I take to represent the static part of whatever module will instantiate `X`. For other module expressions, $\mathsf{Fst}(M)$ is definable in terms of the existing type structure of System F_ω. For example, when M is a structure, $\mathsf{Fst}(M)$ 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, $\mathsf{Fst}(M)$ 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 $\mathsf{Fst}(M)$.`t`, which projects the `t` component from the record of types $\mathsf{Fst}(M)$ 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 $\mathsf{Fst}(M_1)$ and $\mathsf{Fst}(M_2)$ 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, $\mathsf{Fst}(M)$ has kind $\mathsf{Fst}(S)$ 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

[10]Specifically, 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.

[11]When M is a *partial* functor, it has no static part because its body is potentially inseparable. Formally, $\mathsf{Fst}(M)$ 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 $\mathfrak{S}(\mathrm{C})$, 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 $\mathfrak{S}(\mathrm{C})$.

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 $\Sigma\alpha{:}\mathrm{K}_1.\mathrm{K}_2$ 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 $\Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2$ 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 $\Sigma\alpha{:}\mathbf{T}.\mathfrak{S}(\mathtt{int} \times \alpha)$ 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 $\mathsf{Fst}(\mathrm{M})$ as a primitive type constructor, albeit one that is always reducible to some module-free core-language constructor. Treating $\mathsf{Fst}(\mathrm{M})$ 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 $\mathrm{M}_{\mathrm{GUI}}$ 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 $\mathrm{M}_{\mathrm{DICT}}$ 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 $\mathrm{M}_{\mathrm{GUI}}$, and pure but inseparable modules, like $\mathrm{M}_{\mathrm{DICT}}$.

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.

[12] Do 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
Dynamic effects?	Static effects?	in the Present Account
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 $\mathtt{S}$ (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 $\mathfrak{s}_{\mathrm{S}}(\mathrm{M})$ classifies modules of signature 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.

Chapter 3

A Type System for ML Modules: Core Language

In this chapter, I will present the core language of my type system for ML modules. As described in Section 2.2.3, I have lifted my core language almost entirely from the work of Stone and Harper [74] (and also Stone's thesis [72]) on a calculus of singleton kinds. This chapter will therefore be to a large extent a review of Stone and Harper's calculus and its meta-theoretic properties, intended mainly for the reader who is not familiar with their work.

In Section 3.1, I will present the type structure of the core language, *i.e.,* the language of type constructors and kinds, and describe the Stone-Harper algorithm for deciding equivalence of type constructors in the presence of singleton kinds. In Section 3.2, I will present the term structure of the core language and give a type checking algorithm and dynamic semantics for terms.

3.1 Type Constructors and Kinds

3.1.1 Syntax

The syntax of type constructors and kinds is given in Figure 3.1. The language of type constructors (represented by the metavariable C) is a completely standard variant of F_ω.[1] It contains a set of base types b—in particular, unit, product, arrow, universal and existential types—as well as unit, pairing and function operators (and corresponding elimination forms) for building type constructors of higher kind. Constructor variables, written α, are drawn from some countably infinite set *ConVars*. Although a let construct is not included as primitive, it is easily encodable. I will sometimes write "$\mathsf{let}\ \alpha = \mathrm{C}_1\ \mathsf{in}\ \mathrm{C}_2$" as shorthand for $\pi_2\langle \alpha = \mathrm{C}_1, \mathrm{C}_2 \rangle$.[2]

A word about variable bindings: Throughout this thesis, I employ the standard technique of syntactically identifying any two expressions that differ only in their choice of bound variable names. Unless otherwise specified, all the binding constructs I use look like "$\langle x = e_1, e_2 \rangle$", "$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$", or "$x{:}e_1.e_2$", where x is some variable (be it constructor, term or module variable), and the e's are syntactic expressions of some form (be they kinds, constructors, etc.). In all of these

[1] The expressive power of ML only requires us to use the predicative form of F_ω [61], but I have chosen for simplicity to make this language impredicative.

[2] The Stone-Harper calculus only provides a pair construct of the form $\langle \mathrm{C}_1, \mathrm{C}_2 \rangle$. The one I employ here, which binds α to C_1 inside C_2, is not strictly necessary, but I have found it to be rather convenient. In Dreyer *et al.* [11] I verified that Stone and Harper's meta-theory is essentially unchanged when one allows this variant of the pair construct. In a similar vein, Stone and Harper do not include a unit kind either, but supporting it requires only a trivial and obvious extension to their meta-theory (again verified in Dreyer *et al.* [11]).

Constructor Variables	$\alpha, \beta \;\in$	*ConVars*
Kinds	$\mathrm{K}, \mathrm{L} ::=$	$1 \mid \mathbf{T} \mid \mathfrak{S}(\mathrm{C}) \mid \Sigma\alpha{:}\mathrm{K}_1.\mathrm{K}_2 \mid \Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2$
Transparent Kinds	$\mathbb{K}, \mathbb{L} ::=$	$1 \mid \mathfrak{S}(\mathrm{C}) \mid \Sigma\alpha{:}\mathbb{K}_1.\mathbb{K}_2 \mid \Pi\alpha{:}\mathrm{K}_1.\mathbb{K}_2$
Type Constructors	$\mathrm{C}, \mathrm{D} ::=$	$\alpha \mid b \mid \langle\rangle \mid \langle \alpha = \mathrm{C}_1, \mathrm{C}_2 \rangle \mid \pi_i \mathrm{C} \mid \lambda\alpha{:}\mathrm{K}.\mathrm{C} \mid \mathrm{C}_1(\mathrm{C}_2)$
Base Types	$b ::=$	$\mathsf{unit} \mid \mathrm{C}_1 \times \mathrm{C}_2 \mid \mathrm{C}_1 \rightarrow \mathrm{C}_2 \mid \forall\alpha{:}\mathrm{K}.\mathrm{C} \mid \exists\alpha{:}\mathrm{K}.\mathrm{C}$
Static Contexts	$\Delta ::=$	$\emptyset \mid \Delta, \alpha{:}\mathrm{K}$

Figure 3.1: Syntax of Type Constructors and Kinds

cases, x is assumed to be bound in e_2. Thus, for example: In the pair constructor $\langle \alpha = \mathrm{C}_1, \mathrm{C}_2 \rangle$, the variable α stands for the constructor C_1 inside C_2. In the function constructor $\lambda\alpha{:}\mathrm{K}.\mathrm{C}$, the argument variable α, whose kind is K, is bound in the body C. In addition, I will use the notation $e[e_1, \cdots, e_n/x_1, \cdots, x_n]$ to denote the simultaneous capture-avoiding substitution of $e_1, \cdots, e_n$ for $x_1, \cdots, x_n$, respectively, in e.

Now for the interesting part: the kind language. Let us look at the different kind forms one at a time. The unit kind 1 classifies only one type constructor, namely the unit constructor $\langle\rangle$, not to be confused with the *base* type unit. The base kind $\mathbf{T}$, which in many presentations of F_ω is written as $*$ or Ω, classifies those types which themselves classify terms. In a closed program, every type constructor of base kind will be reducible to a syntactic base type b.

The other base kind is the singleton kind $\mathfrak{S}(\mathrm{C})$, a subkind of $\mathbf{T}$ that classifies precisely those type constructors that are equivalent to C. As explained in Section 2.2.3, singletons allow one to support a notion of "type definition" within a regular kind structure. Furthermore, the combination of the "opaque" base kind $\mathbf{T}$ and the "transparent" singleton kind $\mathfrak{S}(\mathrm{C})$ provide a natural basis for translucent signatures in the module language.

The dependent pair kind $\Sigma\alpha{:}\mathrm{K}_1.\mathrm{K}_2$ classifies pairs of constructors whose first component has kind K_1 and whose second component has kind K_2, where α stands for the first component. In other words, a constructor C has kind $\Sigma\alpha{:}\mathrm{K}_1.\mathrm{K}_2$ if and only if $\pi_1\mathrm{C}$ has kind K_1 and $\pi_2\mathrm{C}$ has kind $\mathrm{K}_2[\pi_1\mathrm{C}/\alpha]$. When the bound variable α does not appear free in K_2, I will sometimes write the pair kind as $\mathrm{K}_1 \times \mathrm{K}_2$.

The dependent arrow kind $\Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2$ classifies constructor functions that take an argument of kind K_1 and return a result of kind K_2, where α stands for the argument. Thus, a constructor C has kind $\Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2$ if and only if, whenever D has kind K_1, C(D) has kind $\mathrm{K}_2[\mathrm{D}/\alpha]$. When the bound variable α does not appear free in K_2, I will sometimes write the arrow kind as $\mathrm{K}_1 \rightarrow \mathrm{K}_2$.

While the primitive singleton kind $\mathfrak{S}(\mathrm{C})$ is only valid when C has kind $\mathbf{T}$, singletons at higher kind are in fact definable in terms of the primitive kinds. Figure 3.2 defines a meta-level function $\mathfrak{S}_\mathrm{K}(\mathrm{C})$, which returns a subkind of K classifying precisely those constructors that are equivalent to C (assuming C actually has kind K).[3] At the base kinds $\mathbf{T}$ and $\mathfrak{S}(\mathrm{D})$, $\mathfrak{S}_\mathrm{K}(\mathrm{C})$ is defined as just the primitive singleton $\mathfrak{S}(\mathrm{C})$. The definition of $\mathfrak{S}_\mathrm{K}(\mathrm{C})$ at the remaining kinds reflects the fact that constructor equivalence in the Stone-Harper calculus is *extensional*, *i.e.,* if two constructors are impossible to distinguish by their uses then they are as good as equivalent. Specifically, if we read D's membership in $\mathfrak{S}_\mathrm{K}(\mathrm{C})$ as "D is equivalent to C at kind K," then we see from the definition of $\mathfrak{S}_\mathrm{K}(\mathrm{C})$ that (1) D is equivalent to C at pair kind when $\pi_1\mathrm{D}$ is equivalent to $\pi_1\mathrm{C}$ and $\pi_2\mathrm{D}$ is equivalent to $\pi_2\mathrm{C}$, (2) D is equivalent to C at arrow kind when $\mathrm{D}(\alpha)$ is equivalent to $\mathrm{C}(\alpha)$, and (3)

[3] The function $\mathfrak{S}_\mathrm{K}(\mathrm{C})$ is defined inductively on the *size* of the kind K using the size metric given in Definition 3.1.6.

$$
\begin{array}{lcl}
\mathfrak{S}_1(\mathrm{C}) & \stackrel{\text{def}}{=} & 1 \\
\mathfrak{S}_{\mathbf{T}}(\mathrm{C}) & \stackrel{\text{def}}{=} & \mathfrak{S}(\mathrm{C}) \\
\mathfrak{S}_{\mathfrak{S}(\mathrm{D})}(\mathrm{C}) & \stackrel{\text{def}}{=} & \mathfrak{S}(\mathrm{C}) \\
\mathfrak{S}_{\Sigma\alpha:\mathrm{K}_1.\mathrm{K}_2}(\mathrm{C}) & \stackrel{\text{def}}{=} & \mathfrak{S}_{\mathrm{K}_1}(\pi_1\mathrm{C}) \times \mathfrak{S}_{\mathrm{K}_2[\pi_1\mathrm{C}/\alpha]}(\pi_2\mathrm{C}) \\
\mathfrak{S}_{\Pi\alpha:\mathrm{K}_1.\mathrm{K}_2}(\mathrm{C}) & \stackrel{\text{def}}{=} & \Pi\alpha{:}\mathrm{K}_1.\mathfrak{S}_{\mathrm{K}_2}(\mathrm{C}(\alpha))
\end{array}
$$

Figure 3.2: Singletons at Higher Kinds

$$
\begin{array}{lcl}
\mathsf{Can}(1) & \stackrel{\text{def}}{=} & \langle\rangle \\
\mathsf{Can}(\mathfrak{S}(\mathrm{C})) & \stackrel{\text{def}}{=} & \mathrm{C} \\
\mathsf{Can}(\Sigma\alpha{:}\mathbb{K}_1.\mathbb{K}_2) & \stackrel{\text{def}}{=} & \langle\alpha = \mathsf{Can}(\mathbb{K}_1), \mathsf{Can}(\mathbb{K}_2)\rangle \\
\mathsf{Can}(\Pi\alpha{:}\mathrm{K}_1.\mathbb{K}_2) & \stackrel{\text{def}}{=} & \lambda\alpha{:}\mathrm{K}_1.\mathsf{Can}(\mathbb{K}_2)
\end{array}
$$

Figure 3.3: Canonical Constructors of Transparent Kinds

all constructors of unit kind are equivalent because they are all equally useless.

The definition of higher-order singletons given in Figure 3.2 is not the only possible one. For instance, $\mathfrak{S}_{\mathfrak{S}(\mathrm{D})}(\mathrm{C})$ and $\mathfrak{S}_{\Sigma\alpha:\mathrm{K}_1.\mathrm{K}_2}(\mathrm{C})$ could just as well be defined as $\mathfrak{S}(\mathrm{D})$ and $\Sigma\alpha{:}\mathfrak{S}_{\mathrm{K}_1}(\pi_1\mathrm{C}).\mathfrak{S}_{\mathrm{K}_2}(\pi_2\mathrm{C})$, respectively. It is also likely possible to make $\mathfrak{S}_{\mathrm{K}}(\mathrm{C})$ a primitive kind instead of a macro, and to build in the definitional equations in Figure 3.2 as kind equivalence rules. This would have the slight disadvantage, however, of losing the property that kind equivalence and subtyping (discussed below) are syntax-directed. For simplicity, I have avoided any complications by defining $\mathfrak{S}_{\mathrm{K}}(\mathrm{C})$ precisely as Stone and Harper do.

Higher-order singletons are a special case of a more general subclass of kinds that I call "transparent" and denote by the metavariable $\mathbb{K}$. The syntax of transparent kinds, given in Figure 3.1, requires essentially that no instances of the opaque base kind $\mathbf{T}$ appear on the right-hand side of an arrow. A transparent kind $\mathbb{K}$ has the property that any two constructors of kind $\mathbb{K}$ are equivalent, and in fact transparent kinds are the only kinds to enjoy this property. Consequently, given any transparent kind $\mathbb{K}$, Figure 3.3 shows how to construct a "canonical" constructor $\mathsf{Can}(\mathbb{K})$ of kind $\mathbb{K}$, which is guaranteed to be equivalent to any other inhabitant of $\mathbb{K}$. Transparent kinds will be needed primarily in order to define a notion of transparent signatures in Chapter 4.

Finally, I define a notion of *static contexts*, written Δ, which bind constructor variables to their kinds. I call these contexts static so as to distinguish them from *dynamic contexts* Γ that also bind value and module variables. For all forms of contexts, I require that all variables bound in a context be distinct in order for the context to be syntactically valid. I will also sometimes treat a context as a (meta-level) function from variables to classifiers, *e.g.,* if $\alpha{:}\mathrm{K} \in \Delta$, then $\Delta(\alpha) = \mathrm{K}$.

3.1.2 Static Semantics

Figure 3.4 shows the inference rules for the judgments of well-formed contexts, well-formed kinds, kind equivalence and kind subtyping. The rules for the first three of these judgments are fairly

Well-formed static contexts: $\Delta \vdash \mathrm{ok}$

$$\frac{}{\emptyset \vdash \mathrm{ok}}\ (1) \qquad \frac{\Delta \vdash \mathrm{K\ kind}}{\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{ok}}\ (2)$$

Well-formed kinds: $\Delta \vdash \mathrm{K\ kind}$

$$\frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash 1\ \mathrm{kind}}\ (3) \qquad \frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash \mathbf{T}\ \mathrm{kind}}\ (4) \qquad \frac{\Delta \vdash \mathrm{C} : \mathbf{T}}{\Delta \vdash \mathfrak{s}(\mathrm{C})\ \mathrm{kind}}\ (5)$$

$$\frac{\Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{K}''\ \mathrm{kind}}{\Delta \vdash \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''\ \mathrm{kind}}\ (6) \qquad \frac{\Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{K}''\ \mathrm{kind}}{\Delta \vdash \Pi\alpha{:}\mathrm{K}'.\mathrm{K}''\ \mathrm{kind}}\ (7)$$

Kind equivalence: $\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$

$$\frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash 1 \equiv 1}\ (8) \qquad \frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash \mathbf{T} \equiv \mathbf{T}}\ (9) \qquad \frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Delta \vdash \mathfrak{s}(\mathrm{C}_1) \equiv \mathfrak{s}(\mathrm{C}_2)}\ (10)$$

$$\frac{\Delta \vdash \mathrm{K}_1' \equiv \mathrm{K}_2' \quad \Delta, \alpha{:}\mathrm{K}_1' \vdash \mathrm{K}_1'' \equiv \mathrm{K}_2''}{\Delta \vdash \Sigma\alpha{:}\mathrm{K}_1'.\mathrm{K}_1'' \equiv \Sigma\alpha{:}\mathrm{K}_2'.\mathrm{K}_2''}\ (11) \qquad \frac{\Delta \vdash \mathrm{K}_2' \equiv \mathrm{K}_1' \quad \Delta, \alpha{:}\mathrm{K}_2' \vdash \mathrm{K}_1'' \equiv \mathrm{K}_2''}{\Delta \vdash \Pi\alpha{:}\mathrm{K}_1'.\mathrm{K}_1'' \equiv \Pi\alpha{:}\mathrm{K}_2'.\mathrm{K}_2''}\ (12)$$

Kind subtyping: $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2$

$$\frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash 1 \leq 1}\ (13) \qquad \frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash \mathbf{T} \leq \mathbf{T}}\ (14) \qquad \frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Delta \vdash \mathfrak{s}(\mathrm{C}_1) \leq \mathfrak{s}(\mathrm{C}_2)}\ (15) \qquad \frac{\Delta \vdash \mathrm{C} : \mathbf{T}}{\Delta \vdash \mathfrak{s}(\mathrm{C}) \leq \mathbf{T}}\ (16)$$

$$\frac{\Delta \vdash \mathrm{K}_1' \leq \mathrm{K}_2' \quad \Delta, \alpha{:}\mathrm{K}_1' \vdash \mathrm{K}_1'' \leq \mathrm{K}_2'' \quad \Delta \vdash \Sigma\alpha{:}\mathrm{K}_2'.\mathrm{K}_2''\ \mathrm{kind}}{\Delta \vdash \Sigma\alpha{:}\mathrm{K}_1'.\mathrm{K}_1'' \leq \Sigma\alpha{:}\mathrm{K}_2'.\mathrm{K}_2''}\ (17)$$

$$\frac{\Delta \vdash \mathrm{K}_2' \leq \mathrm{K}_1' \quad \Delta, \alpha{:}\mathrm{K}_2' \vdash \mathrm{K}_1'' \leq \mathrm{K}_2'' \quad \Delta \vdash \Pi\alpha{:}\mathrm{K}_1'.\mathrm{K}_1''\ \mathrm{kind}}{\Delta \vdash \Pi\alpha{:}\mathrm{K}_1'.\mathrm{K}_1'' \leq \Pi\alpha{:}\mathrm{K}_2'.\mathrm{K}_2''}\ (18)$$

Figure 3.4: Inference Rules for Kinds and Static Contexts

obvious. The only point of note is that, following Stone-Harper, for any derivation of a judgment of the form $\Delta \vdash \mathcal{J}$, the static semantics ensures that the well-formedness of the context ($\Delta \vdash \mathrm{ok}$) appears as a subderivation. This explains, for instance, why there is no need for extra premises $\Delta \vdash \mathrm{ok}$ and $\Delta \vdash \mathrm{K}'$ kind in Rules 2 and 7, respectively.

The rules for kind subtyping are also fairly straightforward. For base kinds, subtyping is just equivalence with the addition of Rule 16. This rule allows one to coerce a constructor from the transparent base kind $\mathfrak{s}(\mathrm{C})$ to the opaque base kind $\mathbf{T}$, thereby forgetting that the constructor is equivalent to C. The remaining rules lift this forgetful subtyping to higher kinds, in the standard co- and contra-variant manner. The only point of note here is that, in Rules 17 and 18, the third premise is included in order to ensure Validity (Proposition 3.1.8 below).

Figure 3.5 shows the inference rules for the judgments of well-formed constructors and constructor equivalence. Concerning the former, most of the rules are completely standard for a dependently-typed calculus. The three rules that merit special attention are Rules 31–33. Taken together, these rules enable any type constructor C to be given a most-specific transparent kind

Well-formed constructors: $\Delta \vdash \mathrm{C} : \mathrm{K}$

$$\frac{\Delta \vdash \mathrm{ok} \quad \alpha{:}\mathrm{K} \in \Delta}{\Delta \vdash \alpha : \mathrm{K}}\ (19) \qquad \frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash \mathsf{unit} : \mathbf{T}}\ (20) \qquad \frac{\Delta \vdash \mathrm{C}' : \mathbf{T} \quad \Delta \vdash \mathrm{C}'' : \mathbf{T}}{\Delta \vdash \mathrm{C}' \times \mathrm{C}'' : \mathbf{T}}\ (21)$$

$$\frac{\Delta \vdash \mathrm{C}' : \mathbf{T} \quad \Delta \vdash \mathrm{C}'' : \mathbf{T}}{\Delta \vdash \mathrm{C}' \to \mathrm{C}'' : \mathbf{T}}\ (22) \quad \frac{\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{C} : \mathbf{T}}{\Delta \vdash \forall \alpha{:}\mathrm{K}.\mathrm{C} : \mathbf{T}}\ (23) \quad \frac{\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{C} : \mathbf{T}}{\Delta \vdash \exists \alpha{:}\mathrm{K}.\mathrm{C} : \mathbf{T}}\ (24) \quad \frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash \langle\rangle : 1}\ (25)$$

$$\frac{\Delta \vdash \mathrm{C}' : \mathrm{K}' \quad \Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C}'' : \mathrm{K}''}{\Delta \vdash \langle \alpha = \mathrm{C}', \mathrm{C}'' \rangle : \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''}\ (26) \quad \frac{\Delta \vdash \mathrm{C} : \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''}{\Delta \vdash \pi_1 \mathrm{C} : \mathrm{K}'}\ (27) \quad \frac{\Delta \vdash \mathrm{C} : \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''}{\Delta \vdash \pi_2 \mathrm{C} : \mathrm{K}''[\pi_1 \mathrm{C}/\alpha]}\ (28)$$

$$\frac{\Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C} : \mathrm{K}''}{\Delta \vdash \lambda \alpha{:}\mathrm{K}'.\mathrm{C} : \Pi \alpha{:}\mathrm{K}'.\mathrm{K}''}\ (29) \qquad \frac{\Delta \vdash \mathrm{C} : \Pi \alpha{:}\mathrm{K}'.\mathrm{K}'' \quad \Delta \vdash \mathrm{D} : \mathrm{K}'}{\Delta \vdash \mathrm{C}(\mathrm{D}) : \mathrm{K}''[\mathrm{D}/\alpha]}\ (30)$$

$$\frac{\Delta \vdash \mathrm{C} : \mathbf{T}}{\Delta \vdash \mathrm{C} : \mathfrak{s}(\mathrm{C})}\ (31) \qquad \frac{\Delta \vdash \pi_1 \mathrm{C} : \mathrm{K}' \quad \Delta \vdash \pi_2 \mathrm{C} : \mathrm{K}''}{\Delta \vdash \mathrm{C} : \mathrm{K}' \times \mathrm{K}''}\ (32)$$

$$\frac{\Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C}(\alpha) : \mathrm{K}'' \quad \Delta \vdash \mathrm{C} : \Pi \alpha{:}\mathrm{K}'.\mathrm{L}}{\Delta \vdash \mathrm{C} : \Pi \alpha{:}\mathrm{K}'.\mathrm{K}''}\ (33) \qquad \frac{\Delta \vdash \mathrm{C} : \mathrm{K}' \quad \Delta \vdash \mathrm{K}' \leq \mathrm{K}}{\Delta \vdash \mathrm{C} : \mathrm{K}}\ (34)$$

Constructor equivalence: $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}$

$$\frac{\Delta \vdash \mathrm{C} : \mathrm{K}}{\Delta \vdash \mathrm{C} \equiv \mathrm{C} : \mathrm{K}}\ (35) \qquad \frac{\Delta \vdash \mathrm{C}_2 \equiv \mathrm{C}_1 : \mathrm{K}}{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}}\ (36) \qquad \frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K} \quad \Delta \vdash \mathrm{C}_2 \equiv \mathrm{C}_3 : \mathrm{K}}{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_3 : \mathrm{K}}\ (37)$$

$$\frac{\Delta \vdash \mathrm{C}_1' \equiv \mathrm{C}_2' : \mathbf{T} \quad \Delta \vdash \mathrm{C}_1'' \equiv \mathrm{C}_2'' : \mathbf{T}}{\Delta \vdash \mathrm{C}_1' \times \mathrm{C}_1'' \equiv \mathrm{C}_2' \times \mathrm{C}_2'' : \mathbf{T}}\ (38) \qquad \frac{\Delta \vdash \mathrm{C}_1' \equiv \mathrm{C}_2' : \mathbf{T} \quad \Delta \vdash \mathrm{C}_1'' \equiv \mathrm{C}_2'' : \mathbf{T}}{\Delta \vdash \mathrm{C}_1' \to \mathrm{C}_1'' \equiv \mathrm{C}_2' \to \mathrm{C}_2'' : \mathbf{T}}\ (39)$$

$$\frac{\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2 \quad \Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Delta \vdash \forall \alpha{:}\mathrm{K}_1.\mathrm{C}_1 \equiv \forall \alpha{:}\mathrm{K}_2.\mathrm{C}_2 : \mathbf{T}}\ (40) \qquad \frac{\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2 \quad \Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Delta \vdash \exists \alpha{:}\mathrm{K}_1.\mathrm{C}_1 \equiv \exists \alpha{:}\mathrm{K}_2.\mathrm{C}_2 : \mathbf{T}}\ (41)$$

$$\frac{\Delta \vdash \mathrm{C}_1' \equiv \mathrm{C}_2' : \mathrm{K}' \quad \Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C}_1'' \equiv \mathrm{C}_2'' : \mathrm{K}''}{\Delta \vdash \langle \alpha = \mathrm{C}_1', \mathrm{C}_1'' \rangle \equiv \langle \alpha = \mathrm{C}_2', \mathrm{C}_2'' \rangle : \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''}\ (42)$$

$$\frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''}{\Delta \vdash \pi_1 \mathrm{C}_1 \equiv \pi_1 \mathrm{C}_2 : \mathrm{K}'}\ (43) \qquad \frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''}{\Delta \vdash \pi_2 \mathrm{C}_1 \equiv \pi_2 \mathrm{C}_2 : \mathrm{K}''[\pi_1 \mathrm{C}_1/\alpha]}\ (44)$$

$$\frac{\Delta \vdash \mathrm{K}_1' \equiv \mathrm{K}_2' \quad \Delta, \alpha{:}\mathrm{K}_1' \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}''}{\Delta \vdash \lambda \alpha{:}\mathrm{K}_1'.\mathrm{C}_1 \equiv \lambda \alpha{:}\mathrm{K}_2'.\mathrm{C}_2 : \Pi \alpha{:}\mathrm{K}_1'.\mathrm{K}''}\ (45) \qquad \frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \Pi \alpha{:}\mathrm{K}'.\mathrm{K}'' \quad \Delta \vdash \mathrm{D}_1 \equiv \mathrm{D}_2 : \mathrm{K}'}{\Delta \vdash \mathrm{C}_1(\mathrm{D}_1) \equiv \mathrm{C}_2(\mathrm{D}_2) : \mathrm{K}''[\mathrm{D}_1/\alpha]}\ (46)$$

$$\frac{\Delta \vdash \mathrm{C}_1 : 1 \quad \Delta \vdash \mathrm{C}_2 : 1}{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : 1}\ (47) \qquad \frac{\Delta \vdash \mathrm{C}_1 : \mathfrak{s}(\mathrm{C}) \quad \Delta \vdash \mathrm{C}_2 : \mathfrak{s}(\mathrm{C})}{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathfrak{s}(\mathrm{C})}\ (48)$$

$$\frac{\Delta \vdash \pi_1 \mathrm{C}_1 \equiv \pi_1 \mathrm{C}_2 : \mathrm{K}' \quad \Delta \vdash \pi_2 \mathrm{C}_1 \equiv \pi_2 \mathrm{C}_2 : \mathrm{K}''}{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}' \times \mathrm{K}''}\ (49)$$

$$\frac{\Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C}_1(\alpha) \equiv \mathrm{C}_2(\alpha) : \mathrm{K}'' \quad \Delta \vdash \mathrm{C}_1 : \Pi \alpha{:}\mathrm{K}'.\mathrm{L}_1 \quad \Delta \vdash \mathrm{C}_2 : \Pi \alpha{:}\mathrm{K}'.\mathrm{L}_2}{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \Pi \alpha{:}\mathrm{K}'.\mathrm{K}''}\ (50)$$

$$\frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}' \quad \Delta \vdash \mathrm{K}' \leq \mathrm{K}}{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}}\ (51)$$

Figure 3.5: Inference Rules for Type Constructors

whose only inhabitant is C. This is often referred to as "selfification." Perhaps the easiest way to read the rules is to observe how they allow one to assign to any constructor C the transparent kind $\mathfrak{s}_{\mathrm{K}}(\mathrm{C})$, given that C has kind K to begin with. While it is possible that one could simply replace Rules 31–33 with one rule involving higher-order singletons, I have chosen as before to follow Stone and Harper's presentation.

As for constructor equivalence: Rules 35–37 establish that it is an equivalence relation, a property that is admissible for kinds but must be made explicit for constructors. Rules 38–46 establish that constructor equivalence is a congruence, *i.e.,* constructors that are structurally similar and whose component parts are equivalent are themselves equivalent. Rules 47–48 exhibit that all inhabitants of unit kind 1 are equivalent (to $\langle\rangle$), and similarly all inhabitants of a singleton kind $\mathfrak{s}(\mathrm{C})$ are equivalent (to C). Rules 49–50 ensure that equivalence is extensional, as discussed in the previous section. The extra premises in Rule 50 ensure that α is not free in C_1 and C_2. Finally, Rule 51 enables subsumption for the constructor equivalence judgment.

It is easy to see that constructor equivalence in this calculus, unlike constructor equivalence in System F_ω, is highly dependent on the context in which constructors are compared. In F_ω, two distinct constructor variables α and β will never be considered equivalent, whereas in the presence of singletons α and β may very well be equivalent if, say, they are both bound in the context with kind $\mathfrak{s}(\mathrm{C})$. What is perhaps less obvious is that the equivalence of two constructors also depends on the *kind* at which they are compared, that is, C_1 and C_2 may be equivalent at one kind but not another. The canonical example of this is when $\mathrm{C}_1 = \lambda\alpha{:}\mathbf{T}.\alpha$ and $\mathrm{C}_2 = \lambda\alpha{:}\mathbf{T}.\mathsf{unit}$. When C_1 and C_2 are compared at $\mathbf{T} \to \mathbf{T}$, they are not considered equivalent, as they give different results when applied to any type other than unit. However, when compared at the superkind $\mathfrak{s}(\mathsf{unit}) \to \mathbf{T}$, they are indeed extensionally equivalent: the only valid argument to a function of this superkind is unit, for which C_1 and C_2 return equivalent results. As a consequence, the algorithm presented in Section 3.1.7 for deciding constructor equivalence is both kind- and context-sensitive.

Lastly, the reader may have noticed that the equivalence rules do not include any standard β- or η-equivalence rules. This is because these rules are admissible in the presence of singletons and extensional equivalence. In the case of β-equivalence, this is easy to see by example. Suppose that C and D are constructors of kind $\mathbf{T}$. Then, $\lambda\alpha{:}\mathbf{T}.\mathrm{C}$ can be given kind $\Pi\alpha{:}\mathbf{T}.\mathfrak{s}(\mathrm{C})$. The application rule (Rule 30) tells us then that $(\lambda\alpha{:}\mathbf{T}.\mathrm{C})(\mathrm{D})$ has kind $\mathfrak{s}(\mathrm{C}[\mathrm{D}/\alpha])$ and is therefore equal to $\mathrm{C}[\mathrm{D}/\alpha]$. This reasoning lifts easily to constructors of higher kind (see Proposition 3.1.13). As for η-equivalence, it is essentially just another way of phrasing the concept of extensionality.

3.1.3 Basic Structural Properties

In this section I state several basic structural properties concerning the language of constructors and kinds described above. Throughout I will write $\Delta \vdash \mathcal{J}$ to denote any judgment with right hand side $\mathcal{J}$, and $\mathrm{FV}(\mathcal{J})$ to denote the set of variables that appear free in one or more syntactic components of $\mathcal{J}$. In addition, I use the "=" sign to indicate syntactic equality (modulo α-equivalence).

Proposition 3.1.1 (Subderivations)

1. Every proof of $\Delta \vdash \mathcal{J}$ contains a subderivation of $\Delta \vdash \mathrm{ok}$.

2. Every proof of $\Delta_1, \alpha{:}\mathrm{K}, \Delta_2 \vdash \mathcal{J}$ contains a strict subderivation of $\Delta_1 \vdash \mathrm{K}\ \mathrm{kind}$.

Proposition 3.1.2 (Free Variable Containment)
If $\Delta \vdash \mathcal{J}$, then $\mathrm{FV}(\mathcal{J}) \subseteq \mathrm{dom}(\Delta)$.

- $\Delta' \vdash \gamma : \Delta$ iff
  1. $\Delta' \vdash \mathrm{ok}$
  2. $\forall \alpha{:}\mathrm{K} \in \mathrm{dom}(\Delta).\ \Delta' \vdash \gamma(\alpha) : \gamma(\mathrm{K})$
- $\Delta' \vdash \gamma_1 \equiv \gamma_2 : \Delta$ iff
  1. $\Delta' \vdash \gamma_1 : \Delta$ and $\Delta' \vdash \gamma_2 : \Delta$
  2. $\forall \alpha{:}\mathrm{K} \in \mathrm{dom}(\Delta).\ \Delta' \vdash \gamma_1(\alpha) \equiv \gamma_2(\alpha) : \gamma_1(\mathrm{K})$

Figure 3.6: Typing and Equivalence Judgments for Static Substitutions

Definition 3.1.3 (Context Extension)
The context Δ_2 is defined to *extend* the context Δ_1 (written $\Delta_2 \supseteq \Delta_1$) if the contexts viewed as partial functions give the same result for every variable bound in $\mathrm{dom}(\Delta_1)$. (Note that this is a purely syntactic condition and does not imply that either context is well-formed.)

Proposition 3.1.4 (Weakening)
1. If $\Delta_1 \vdash \mathcal{J}$, $\Delta_2 \supseteq \Delta_1$, and $\Delta_2 \vdash \mathrm{ok}$, then $\Delta_2 \vdash \mathcal{J}$.

2. If $\Delta_1, \alpha{:}\mathrm{K}_2, \Delta_2 \vdash \mathcal{J}$, $\Delta_1 \vdash \mathrm{K}_1 \leq \mathrm{K}_2$ and $\Delta_1 \vdash \mathrm{K}_1$ kind, then $\Delta_1, \alpha{:}\mathrm{K}_1, \Delta_2 \vdash \mathcal{J}$.

Static substitutions γ are defined as maps from constructor variables to constructors, which may be applied (in the usual capture-avoiding manner) to arbitrary syntactic expressions. We denote the identity substitution as **id** and substitution extension as $\gamma[\alpha \mapsto \mathrm{C}]$. Figure 3.6 defines typing and equivalence judgments for substitutions.

Proposition 3.1.5 (Substitution)
1. If $\Delta \vdash \mathcal{J}$ and $\Delta' \vdash \gamma : \Delta$, then $\Delta' \vdash \gamma(\mathcal{J})$.

2. If $\Delta_1, \alpha{:}\mathrm{K}, \Delta_2 \vdash \mathcal{J}$ and $\Delta_1 \vdash \mathrm{C} : \mathrm{K}$, then $\Delta_1, \Delta_2[\mathrm{C}/\alpha] \vdash \mathcal{J}[\mathrm{C}/\alpha]$.

3.1.4 Other Declarative Properties

In this section I state several other useful declarative properties, the most important being Validity and Functionality. Validity ensures that all constructors or kinds mentioned inside a derivable judgment are well-formed. Functionality ensures that applying equivalent substitutions to two sides of a $\equiv$ or $\leq$ judgment does not affect the derivability of the judgment. I also establish that kind equivalence is an equivalence relation and kind subtyping is a partial order.

First, it is useful for purposes of induction to have a measure of the "size" of a kind that is invariant under substitution, *i.e.,* such that $\mathrm{size}(\mathrm{K}) = \mathrm{size}(\gamma\mathrm{K})$ for any γ and K.

Definition 3.1.6 (Sizes of Kinds)
Let the *size* of a kind K, written size(K), be defined inductively as follows:

$$
\begin{array}{lcl}
\mathrm{size}(1) & \stackrel{\mathrm{def}}{=} & 1 \\
\mathrm{size}(\mathbf{T}) & \stackrel{\mathrm{def}}{=} & 1 \\
\mathrm{size}(\mathfrak{S}(\mathrm{C})) & \stackrel{\mathrm{def}}{=} & 2 \\
\mathrm{size}(\Sigma\alpha{:}\mathrm{K}_1.\mathrm{K}_2) & \stackrel{\mathrm{def}}{=} & 1 + \mathrm{size}(\mathrm{K}_1) + \mathrm{size}(\mathrm{K}_2) \\
\mathrm{size}(\Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2) & \stackrel{\mathrm{def}}{=} & 1 + \mathrm{size}(\mathrm{K}_1) + \mathrm{size}(\mathrm{K}_2)
\end{array}
$$

Proposition 3.1.7 (Reflexivity)
If $\Delta \vdash \mathrm{K}$ kind, then $\Delta \vdash \mathrm{K} \equiv \mathrm{K}$ and $\Delta \vdash \mathrm{K} \leq \mathrm{K}$.

Proposition 3.1.8 (Validity)

1. If $\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$, then $\Delta \vdash \mathrm{K}_1$ kind and $\Delta \vdash \mathrm{K}_2$ kind.
2. If $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2$, then $\Delta \vdash \mathrm{K}_1$ kind and $\Delta \vdash \mathrm{K}_2$ kind.
3. If $\Delta \vdash \mathrm{C} : \mathrm{K}$, then $\Delta \vdash \mathrm{K}$ kind.
4. If $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}$, then $\Delta \vdash \mathrm{C}_1 : \mathrm{K}$, $\Delta \vdash \mathrm{C}_2 : \mathrm{K}$, and $\Delta \vdash \mathrm{K}$ kind.

Proposition 3.1.9 (Symmetry and Transitivity of Kind Equivalence)

1. If $\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$, then $\Delta \vdash \mathrm{K}_2 \equiv \mathrm{K}_1$.
2. If $\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$ and $\Delta \vdash \mathrm{K}_2 \equiv \mathrm{K}_3$, then $\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_3$.

Proposition 3.1.10 (Antisymmetry and Transitivity of Kind Subtyping)

1. $\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$ if and only if $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2$ and $\Delta \vdash \mathrm{K}_2 \leq \mathrm{K}_1$.
2. If $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2$ and $\Delta \vdash \mathrm{K}_2 \leq \mathrm{K}_3$, then $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_3$.

Proposition 3.1.11 (Functionality)
Suppose $\Delta' \vdash \gamma_1 \equiv \gamma_2 : \Delta$.

1. If $\Delta \vdash \mathrm{K}$ kind, then $\Delta' \vdash \gamma_1\mathrm{K} \equiv \gamma_2\mathrm{K}$.
2. If $\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$, then $\Delta' \vdash \gamma_1\mathrm{K}_1 \equiv \gamma_2\mathrm{K}_2$.
3. If $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2$, then $\Delta' \vdash \gamma_1\mathrm{K}_1 \leq \gamma_2\mathrm{K}_2$.
4. If $\Delta \vdash \mathrm{C} : \mathrm{K}$, then $\Delta' \vdash \gamma_1\mathrm{C} \equiv \gamma_2\mathrm{C} : \gamma_1\mathrm{K}$.
5. If $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}$, then $\Delta' \vdash \gamma_1\mathrm{C}_1 \equiv \gamma_2\mathrm{C}_2 : \gamma_1\mathrm{K}$.

3.1.5 Admissible Rules

Here I enumerate some important admissible rules, which fall into three categories. First, Proposition 3.1.12 states that the inference rules involving singleton kinds extend to higher-order singletons (as defined in Figure 3.2). Since the well-formedness of $\mathfrak{S}_{\mathrm{K}}(\mathrm{C})$ does not necessarily imply that C has kind K, the latter must be added as a premise to the higher-order variants of some of the rules.

Second, Proposition 3.1.13 states that β- and η-equivalence rules for functions and products are admissible, and also gives an alternative formulation of the typing, equivalence and extensionality rules for products. The notation $\langle \mathrm{C}_1, \mathrm{C}_2 \rangle$ is shorthand for $\langle \alpha = \mathrm{C}_1, \mathrm{C}_2 \rangle$, where $\alpha \notin \mathrm{FV}(\mathrm{C}_2)$.

Third, Proposition 3.1.14 gives several properties of transparent kinds, the most important of which is that any two constructors of a transparent kind are equivalent *at that kind*. As discussed in Section 3.1.2, this does not imply that the constructors are equivalent at all kinds. I give a proof only for this third proposition, as proofs for the first two can be found in Stone's thesis [72].

Proposition 3.1.12 (Higher-Order Singleton Rules)

1. $\gamma(\mathfrak{s}_{\mathrm{K}}(\mathrm{C})) = \mathfrak{s}_{\gamma\mathrm{K}}(\gamma\mathrm{C})$.
2. If $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}$, then $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathfrak{s}_{\mathrm{K}}(\mathrm{C}_2)$.
3. If $\Delta \vdash \mathrm{C} : \mathrm{K}$, then $\Delta \vdash \mathfrak{s}_{\mathrm{K}}(\mathrm{C})$ kind and $\Delta \vdash \mathrm{C} : \mathfrak{s}_{\mathrm{K}}(\mathrm{C})$.
4. If $\Delta \vdash \mathrm{C}_1 : \mathfrak{s}_{\mathrm{K}}(\mathrm{C}_2)$ and $\Delta \vdash \mathrm{C}_2 : \mathrm{K}$, then $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathfrak{s}_{\mathrm{K}}(\mathrm{C}_2)$.
5. If $\Delta \vdash \mathrm{C} : \mathrm{K}$, then $\Delta \vdash \mathfrak{s}_{\mathrm{K}}(\mathrm{C}) \leq \mathrm{K}$.
6. If $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}_1$ and $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2$, then $\Delta \vdash \mathfrak{s}_{\mathrm{K}_1}(\mathrm{C}_1) \leq \mathfrak{s}_{\mathrm{K}_2}(\mathrm{C}_2)$.

Proposition 3.1.13 (Admissibility of Beta, Eta, and Alternative Product Rules)

1. If $\Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C} : \mathrm{K}''$ and $\Delta \vdash \mathrm{C}' : \mathrm{K}'$, then $\Delta \vdash (\lambda\alpha{:}\mathrm{K}'.\mathrm{C})(\mathrm{C}') \equiv \mathrm{C}[\mathrm{C}'/\alpha] : \mathrm{K}''[\mathrm{C}'/\alpha]$.
2. If $\Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}''$ and $\Delta \vdash \mathrm{C}_1' \equiv \mathrm{C}_2' : \mathrm{K}'$,
   then $\Delta \vdash (\lambda\alpha{:}\mathrm{K}'.\mathrm{C}_1)(\mathrm{C}_1') \equiv \mathrm{C}_2[\mathrm{C}_2'/\alpha] : \mathrm{K}''[\mathrm{C}_1'/\alpha]$.
3. If $\Delta \vdash \mathrm{C}_1 : \mathrm{K}_1$ and $\Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}_2 : \mathrm{K}_2$, then $\Delta \vdash \pi_1\langle\alpha = \mathrm{C}_1, \mathrm{C}_2\rangle \equiv \mathrm{C}_1 : \mathrm{K}_1$
   and $\Delta \vdash \pi_2\langle\alpha = \mathrm{C}_1, \mathrm{C}_2\rangle \equiv \mathrm{C}_2[\mathrm{C}_1/\alpha] : \mathrm{K}_2[\mathrm{C}_1/\alpha]$.
4. If $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_1' : \mathrm{K}_1$ and $\Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}_2 \equiv \mathrm{C}_2' : \mathrm{K}_2$, then $\Delta \vdash \pi_1\langle\alpha = \mathrm{C}_1, \mathrm{C}_2\rangle \equiv \mathrm{C}_1' : \mathrm{K}_1$
   and $\Delta \vdash \pi_2\langle\alpha = \mathrm{C}_1, \mathrm{C}_2\rangle \equiv \mathrm{C}_2'[\mathrm{C}_1'/\alpha] : \mathrm{K}_2[\mathrm{C}_1'/\alpha]$.
5. If $\Delta \vdash \mathrm{C} : \Pi\alpha{:}\mathrm{K}'.\mathrm{K}''$, then $\Delta \vdash \mathrm{C} \equiv \lambda\alpha{:}\mathrm{K}'.\mathrm{C}(\alpha) : \Pi\alpha{:}\mathrm{K}'.\mathrm{K}''$.
6. If $\Delta \vdash \mathrm{C} : \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''$, then $\Delta \vdash \mathrm{C} \equiv \langle\pi_1\mathrm{C}, \pi_2\mathrm{C}\rangle : \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''$.
7. If $\Delta \vdash \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''$ kind, $\Delta \vdash \mathrm{C}' : \mathrm{K}'$, and $\Delta \vdash \mathrm{C}'' : \mathrm{K}''[\mathrm{C}'/\alpha]$, then $\Delta \vdash \langle\mathrm{C}', \mathrm{C}''\rangle : \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''$.
8. If $\Delta \vdash \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''$ kind, $\Delta \vdash \mathrm{C}_1' \equiv \mathrm{C}_2' : \mathrm{K}'$, and $\Delta \vdash \mathrm{C}_1'' \equiv \mathrm{C}_2'' : \mathrm{K}''[\mathrm{C}_1'/\alpha]$,
   then $\Delta \vdash \langle\mathrm{C}_1', \mathrm{C}_1''\rangle \equiv \langle\mathrm{C}_2', \mathrm{C}_2''\rangle : \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''$.
9. If $\Delta \vdash \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''$ kind, $\Delta \vdash \pi_1\mathrm{C}_1 \equiv \pi_1\mathrm{C}_2 : \mathrm{K}'$, and $\Delta \vdash \pi_2\mathrm{C}_1 \equiv \pi_2\mathrm{C}_2 : \mathrm{K}''[\pi_1\mathrm{C}_1/\alpha]$,
   then $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}''$.

Proposition 3.1.14 (Properties of Transparent Kinds)

1. If $\Delta \vdash \mathrm{C} : \mathbb{K}$, then $\Delta \vdash \mathfrak{s}_{\mathbb{K}}(\mathrm{C}) \equiv \mathbb{K}$.
2. If $\Delta \vdash \mathrm{C}_1 : \mathbb{K}$ and $\Delta \vdash \mathrm{C}_2 : \mathbb{K}$, then $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbb{K}$.
3. If $\Delta \vdash \mathrm{K}' \leq \mathbb{K}$, then K' is transparent.
4. $\mathfrak{s}_{\mathrm{K}}(\mathrm{C})$ is transparent.
5. If $\Delta \vdash \mathbb{K}$ kind, then $\Delta \vdash \mathsf{Can}(\mathbb{K}) : \mathbb{K}$.

Proof:

1. By induction on the size of $\mathbb{K}$.
   - Case: $\mathbb{K} = 1$. Trivial.
   - Case: $\mathbb{K} = \mathfrak{s}(\mathrm{D})$. Since $\Delta \vdash \mathrm{C} : \mathfrak{s}(\mathrm{D})$, we have $\Delta \vdash \mathrm{C} \equiv \mathrm{D} : \mathbf{T}$, and thus $\Delta \vdash \mathfrak{s}(\mathrm{C}) \equiv \mathfrak{s}(\mathrm{D})$.
   - Case: $\mathbb{K} = \Sigma\alpha{:}\mathbb{K}_1.\mathbb{K}_2$.
     (a) Since $\Delta \vdash \pi_1\mathrm{C} : \mathbb{K}_1$, by induction $\Delta \vdash \mathfrak{s}_{\mathbb{K}_1}(\pi_1\mathrm{C}) \equiv \mathbb{K}_1$.
     (b) Since $\Delta \vdash \pi_2\mathrm{C} : \mathbb{K}_2[\pi_1\mathrm{C}/\alpha]$, by induction $\Delta \vdash \mathfrak{s}_{\mathbb{K}_2[\pi_1\mathrm{C}/\alpha]}(\pi_2\mathrm{C}) \equiv \mathbb{K}_2[\pi_1\mathrm{C}/\alpha]$.
     (c) By Proposition 3.1.12, $\Delta, \alpha{:}\mathfrak{s}_{\mathbb{K}_1}(\pi_1\mathrm{C}) \vdash \pi_1\mathrm{C} \equiv \alpha : \mathfrak{s}_{\mathbb{K}_1}(\pi_1\mathrm{C})$.
     (d) By Functionality, $\Delta, \alpha{:}\mathfrak{s}_{\mathbb{K}_1}(\pi_1\mathrm{C}) \vdash \mathbb{K}_2[\pi_1\mathrm{C}/\alpha] \equiv \mathbb{K}_2$.
     (e) Thus, $\Delta \vdash \mathfrak{s}_{\mathbb{K}_1}(\pi_1\mathrm{C}) \times \mathfrak{s}_{\mathbb{K}_2[\pi_1\mathrm{C}/\alpha]}(\pi_2\mathrm{C}) \equiv \Sigma\alpha{:}\mathbb{K}_1.\mathbb{K}_2$.
   - Case: $\mathbb{K} = \Pi\alpha{:}\mathrm{K}_1.\mathbb{K}_2$.
     (a) Since $\Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}(\alpha) : \mathbb{K}_2$, by induction $\Delta, \alpha{:}\mathrm{K}_1 \vdash \mathfrak{s}_{\mathbb{K}_2}(\mathrm{C}(\alpha)) \equiv \mathbb{K}_2$.
     (b) Thus, $\Delta \vdash \Pi\alpha{:}\mathrm{K}_1.\mathfrak{s}_{\mathbb{K}_2}(\mathrm{C}(\alpha)) \equiv \Pi\alpha{:}\mathrm{K}_1.\mathbb{K}_2$.

2. (a) By Part 1, $\Delta \vdash \mathfrak{s}_{\mathbb{K}}(\mathrm{C}_1) \equiv \mathbb{K}$ and $\Delta \vdash \mathfrak{s}_{\mathbb{K}}(\mathrm{C}_2) \equiv \mathbb{K}$, and thus $\Delta \vdash \mathfrak{s}_{\mathbb{K}}(\mathrm{C}_1) \equiv \mathfrak{s}_{\mathbb{K}}(\mathrm{C}_2)$.
   (b) By Part 3 of Proposition 3.1.12, $\Delta \vdash \mathrm{C}_1 : \mathfrak{s}_{\mathbb{K}}(\mathrm{C}_1)$, and thus $\Delta \vdash \mathrm{C}_1 : \mathfrak{s}_{\mathbb{K}}(\mathrm{C}_2)$.
   (c) By Part 4 of Proposition 3.1.12, $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathfrak{s}_{\mathbb{K}}(\mathrm{C}_2)$, and thus $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbb{K}$.

3–5. Straightforward.

■

3.1.6 Kind Checking and Synthesis

Figure 3.7 shows Stone and Harper's algorithm for synthesizing the principal (most-precise) kind for a given type constructor. Checking whether a constructor has a certain kind is then simply a matter of checking whether the constructor's principal kind is a subtype of it. The algorithm itself follows the typing rules for constructors fairly closely, except that it only performs selfification in the variable and base type cases, and it only uses subsumption when checking a function argument against the domain kind of the function. In addition, unlike the declarative rules, the algorithm requires as a precondition that the context it is given is well-formed. Here I state several properties of kind synthesis, including that it is sound, complete and deterministic:

Proposition 3.1.15 (Soundness and Other Properties of Kind Checking/Synthesis) Assume $\Delta \vdash \mathrm{ok}$.

1. If $\Delta \vdash \mathrm{C} \Rightarrow \mathrm{K}$ or $\Delta \vdash \mathrm{C} \Leftarrow \mathrm{K}$, then $\Delta \vdash \mathrm{C} : \mathrm{K}$.

2. Synthesis is deterministic, *i.e.,* if $\Delta \vdash \mathrm{C} \Rightarrow \mathrm{K}_1$ and $\Delta \vdash \mathrm{C} \Rightarrow \mathrm{K}_2$, then $\mathrm{K}_1 = \mathrm{K}_2$.

3. If $\Delta \vdash \mathrm{C} \Rightarrow \mathrm{K}$, then K is transparent.

4. For $\mathcal{J}$ ranging over any judgment defined in Figure 3.7, if $\Delta \vdash \mathcal{J}$ and $\Delta' \supseteq \Delta$ and $\Delta' \vdash \mathrm{ok}$, then $\Delta' \vdash \mathcal{J}$.

Kind checking: $\Delta \vdash \mathrm{C} \Leftarrow \mathrm{K}$

$$\frac{\Delta \vdash \mathrm{C} \Rightarrow \mathrm{K}' \quad \Delta \vdash \mathrm{K}' \leq \mathrm{K}}{\Delta \vdash \mathrm{C} \Leftarrow \mathrm{K}}$$

Base type well-formedness: $\Delta \vdash b$ ok

$$\frac{}{\Delta \vdash \mathsf{unit}\ \mathrm{ok}} \qquad \frac{\Delta \vdash \mathrm{C}_1 \Leftarrow \mathbf{T} \quad \Delta \vdash \mathrm{C}_2 \Leftarrow \mathbf{T}}{\Delta \vdash \mathrm{C}_1 \times \mathrm{C}_2\ \mathrm{ok}} \qquad \frac{\Delta \vdash \mathrm{C}_1 \Leftarrow \mathbf{T} \quad \Delta \vdash \mathrm{C}_2 \Leftarrow \mathbf{T}}{\Delta \vdash \mathrm{C}_1 \rightarrow \mathrm{C}_2\ \mathrm{ok}}$$

$$\frac{\Delta \vdash \mathrm{K}\ \mathrm{kind} \quad \Delta, \alpha{:}\mathrm{K} \vdash \mathrm{C} \Leftarrow \mathbf{T}}{\Delta \vdash \forall \alpha{:}\mathrm{K}.\mathrm{C}\ \mathrm{ok}} \qquad \frac{\Delta \vdash \mathrm{K}\ \mathrm{kind} \quad \Delta, \alpha{:}\mathrm{K} \vdash \mathrm{C} \Leftarrow \mathbf{T}}{\Delta \vdash \exists \alpha{:}\mathrm{K}.\mathrm{C}\ \mathrm{ok}}$$

Principal kind synthesis: $\Delta \vdash \mathrm{C} \Rightarrow \mathrm{K}$

$$\frac{\alpha{:}\mathrm{K} \in \Delta}{\Delta \vdash \alpha \Rightarrow \mathfrak{s}_{\mathrm{K}}(\alpha)} \qquad \frac{\Delta \vdash b\ \mathrm{ok}}{\Delta \vdash b \Rightarrow \mathfrak{s}(b)}$$

$$\frac{\Delta \vdash \mathrm{C}' \Rightarrow \mathrm{K}' \quad \Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C}'' \Rightarrow \mathrm{K}''}{\Delta \vdash \langle \alpha = \mathrm{C}', \mathrm{C}'' \rangle \Rightarrow \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''} \qquad \frac{\Delta \vdash \mathrm{C} \Rightarrow \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''}{\Delta \vdash \pi_1 \mathrm{C} \Rightarrow \mathrm{K}'} \qquad \frac{\Delta \vdash \mathrm{C} \Rightarrow \Sigma \alpha{:}\mathrm{K}'.\mathrm{K}''}{\Delta \vdash \pi_2 \mathrm{C} \Rightarrow \mathrm{K}''[\pi_1 \mathrm{C}/\alpha]}$$

$$\frac{\Delta \vdash \mathrm{K}'\ \mathrm{kind} \quad \Delta, \alpha{:}\mathrm{K}' \vdash \mathrm{C} \Rightarrow \mathrm{K}''}{\Delta \vdash \lambda \alpha{:}\mathrm{K}'.\mathrm{C} \Rightarrow \Pi \alpha{:}\mathrm{K}'.\mathrm{K}''} \qquad \frac{\Delta \vdash \mathrm{C} \Rightarrow \Pi \alpha{:}\mathrm{K}'.\mathrm{K}'' \quad \Delta \vdash \mathrm{D} \Leftarrow \mathrm{K}'}{\Delta \vdash \mathrm{C}(\mathrm{D}) \Rightarrow \mathrm{K}''[\mathrm{D}/\alpha]}$$

Figure 3.7: Kind Checking and Principal Kind Synthesis

Proposition 3.1.16 (Completeness of Kind Checking)
If $\Delta \vdash \mathrm{C} : \mathrm{K}$, then $\Delta \vdash \mathrm{C} \Leftarrow \mathfrak{s}_{\mathrm{K}}(\mathrm{C})$ (and therefore $\Delta \vdash \mathrm{C} \Leftarrow \mathrm{K}$ as well).

As the kind checking algorithm is syntax-directed, showing that it terminates reduces to finding decision procedures for the kind well-formedness and subtyping judgments. Both of these are syntax-directed as well, but the latter requires a method of deciding constructor equivalence in the case of Rule 15. The proof that such a method exists is quite difficult and constitutes Stone and Harper's chief contribution. I discuss their algorithm and proof of correctness in the next section.

3.1.7 Deciding Constructor Equivalence

The Stone-Harper algorithm for deciding constructor equivalence is shown in Figures 3.8 and 3.9. It comprises a number of interlocking judgments, on which I will attempt now to impose some narrative structure.

First of all, suppose we are given two well-formed constructors C_1 and C_2 to be compared at kind K in context Δ. The main judgment $\Delta \vdash \mathrm{C}_1 \Leftrightarrow \mathrm{C}_2 : \mathrm{K}$ determines whether they are equivalent by dividing the problem into a series of subproblems at base kinds. This makes sense due to extensionality: C_1 and C_2 are equivalent at pair kind precisely when their first projections are equivalent and their second projections are equivalent, and they are equivalent at arrow kind precisely when, for any argument α of the domain kind, $\mathrm{C}_1(\alpha)$ is equivalent at $\mathrm{C}_2(\alpha)$ at the result kind. At the unit and singleton base kinds, the algorithm trivially returns a positive answer because all constructors are equivalent at one of those kinds. At kind **T**, however, we must actually look at the constructors!

$$\begin{array}{lrcl} \text{Elimination Contexts} & \mathcal{E} & ::= & \bullet \mid \mathcal{E}(\mathrm{C}) \mid \pi_i\mathcal{E} \\ \text{Constructor Paths} & \mathrm{P} & ::= & b \mid \mathcal{E}\{\alpha\} \end{array}$$

$\boxed{\textbf{Natural kind extraction: } \Delta \vdash \mathrm{P} \uparrow \mathrm{K}}$

$$\begin{array}{ll} \Delta \vdash b \uparrow \mathbf{T} & \\ \Delta \vdash \alpha \uparrow \Delta(\alpha) & \\ \Delta \vdash \mathrm{P}(\mathrm{C}) \uparrow \mathrm{K}''[\mathrm{C}/\alpha] & \text{if } \Delta \vdash \mathrm{P} \uparrow \Pi\alpha{:}\mathrm{K}'.\mathrm{K}'' \\ \Delta \vdash \pi_1\mathrm{P} \uparrow \mathrm{K}' & \text{if } \Delta \vdash \mathrm{P} \uparrow \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}'' \\ \Delta \vdash \pi_2\mathrm{P} \uparrow \mathrm{K}''[\pi_1\mathrm{P}/\alpha] & \text{if } \Delta \vdash \mathrm{P} \uparrow \Sigma\alpha{:}\mathrm{K}'.\mathrm{K}'' \end{array}$$

$\boxed{\textbf{Weak head reduction: } \Delta \vdash \mathrm{C}_1 \xrightarrow{\text{wh}} \mathrm{C}_2}$

$$\begin{array}{ll} \Delta \vdash \mathcal{E}\{(\lambda\alpha{:}\mathrm{K}'.\mathrm{C})\mathrm{C}'\} \xrightarrow{\text{wh}} \mathcal{E}\{\mathrm{C}[\mathrm{C}'/\alpha]\} & \\ \Delta \vdash \mathcal{E}\{\pi_1\langle\alpha = \mathrm{C}', \mathrm{C}''\rangle\} \xrightarrow{\text{wh}} \mathcal{E}\{\mathrm{C}'\} & \\ \Delta \vdash \mathcal{E}\{\pi_2\langle\alpha = \mathrm{C}', \mathrm{C}''\rangle\} \xrightarrow{\text{wh}} \mathcal{E}\{\mathrm{C}''[\mathrm{C}'/\alpha]\} & \\ \Delta \vdash \mathrm{P} \xrightarrow{\text{wh}} \mathrm{C} & \text{if } \Delta \vdash \mathrm{P} \uparrow \mathfrak{s}(\mathrm{C}) \end{array}$$

$\boxed{\textbf{Weak head normalization: } \Delta \vdash \mathrm{C} \overset{\text{wh}}{\Longrightarrow} \mathrm{D}}$

$$\begin{array}{ll} \Delta \vdash \mathrm{C} \overset{\text{wh}}{\Longrightarrow} \mathrm{D} & \text{if } \Delta \vdash \mathrm{C} \xrightarrow{\text{wh}} \mathrm{C}' \text{ and } \Delta \vdash \mathrm{C}' \overset{\text{wh}}{\Longrightarrow} \mathrm{D} \\ \Delta \vdash \mathrm{C} \overset{\text{wh}}{\Longrightarrow} \mathrm{C} & \text{otherwise} \end{array}$$

Figure 3.8: Weak Head Normalization for Type Constructors

When comparing two constructors at kind $\mathbf{T}$, the algorithm first reduces the constructors to weak head normal form (WHNF) [62]. Since the constructors have kind $\mathbf{T}$, their WHNF's will not be λ-abstractions, but rather *paths*, whose syntax is described at the top of Figure 3.8. A path P is either a base type or a sequence of eliminations (*i.e.,* projections and applications) rooted at a constructor variable. The notation $\mathcal{E}\{\mathrm{C}\}$ used in the definition of paths in Figure 3.8 signifies the substitution of C into the single hole $\bullet$ in the elimination context $\mathcal{E}$.

The first three rules in the weak head reduction judgment $\Delta \vdash \mathrm{C}_1 \xrightarrow{\text{wh}} \mathrm{C}_2$ are completely standard β-reduction. The fourth rule is non-standard—it says that being a path is not equivalent to being in WHNF; to be in WHNF a path must also be *abstract*. For example, if α is bound in the context with kind $\mathbf{T}$, then α is an abstract type. If α is bound with $\mathfrak{s}(\mathrm{C})$, however, then α is transparently equal to C and may thus be reduced to it. One can think of this reduction step as "looking up the definition of a type variable." Whether a path has a "definition" or not is determined by a judgment called "natural kind extraction" and written $\Delta \vdash \mathrm{C} \uparrow \mathrm{K}$. Intuitively, the natural kind of a constructor is the kind you would synthesize for it if the selfification rules did not exist. This intuition is reflected in the following fact, connecting principal and natural kinds.

Proposition 3.1.17 (Connection Between Natural and Principal Kinds)
If $\Delta \vdash \mathrm{P} \Rightarrow \mathrm{K}$, then $\Delta \vdash \mathrm{P} \uparrow \mathrm{L}$, $\Delta \vdash \mathrm{P} : \mathrm{L}$, and $\mathrm{K} = \mathfrak{s}_{\mathrm{L}}(\mathrm{P})$.

The natural kind of a path will be of the form $\mathfrak{s}(\mathrm{C})$ if and only if it has a definition in the context

Algorithmic kind equivalence: $\Delta \vdash K_1 \Leftrightarrow K_2$

$\Delta \vdash 1 \Leftrightarrow 1$	
$\Delta \vdash \mathbf{T} \Leftrightarrow \mathbf{T}$	
$\Delta \vdash \mathfrak{s}(C_1) \Leftrightarrow \mathfrak{s}(C_2)$	if $\Delta \vdash C_1 \Leftrightarrow C_2 : \mathbf{T}$
$\Delta \vdash \Pi\alpha{:}K_1'.K_1'' \Leftrightarrow \Pi\alpha{:}K_2'.K_2''$	if $\Delta \vdash K_1' \Leftrightarrow K_2'$ and $\Delta, \alpha{:}K_1' \vdash K_1'' \Leftrightarrow K_2''$
$\Delta \vdash \Sigma\alpha{:}K_1'.K_1'' \Leftrightarrow \Sigma\alpha{:}K_2'.K_2''$	if $\Delta \vdash K_1' \Leftrightarrow K_2'$ and $\Delta, \alpha{:}K_1' \vdash K_1'' \Leftrightarrow K_2''$

Algorithmic constructor equivalence: $\Delta \vdash C_1 \Leftrightarrow C_2 : K$

$\Delta \vdash C_1 \Leftrightarrow C_2 : 1$	
$\Delta \vdash C_1 \Leftrightarrow C_2 : \mathfrak{s}(C)$	
$\Delta \vdash C_1 \Leftrightarrow C_2 : \mathbf{T}$	if $\Delta \vdash C_1 \stackrel{\text{wh}}{\Longrightarrow} P_1$, $\Delta \vdash C_2 \stackrel{\text{wh}}{\Longrightarrow} P_2$, and $\Delta \vdash P_1 \leftrightarrow P_2 \uparrow \mathbf{T}$
$\Delta \vdash C_1 \Leftrightarrow C_2 : \Pi\alpha{:}K'.K''$	if $\Delta, \alpha{:}K' \vdash C_1(\alpha) \Leftrightarrow C_2(\alpha) : K''$
$\Delta \vdash C_1 \Leftrightarrow C_2 : \Sigma\alpha{:}K'.K''$	if $\Delta \vdash \pi_1 C_1 \Leftrightarrow \pi_1 C_2 : K'$ and $\Delta \vdash \pi_2 C_1 \Leftrightarrow \pi_2 C_2 : K''[\pi_1 C_1/\alpha]$

Algorithmic path equivalence: $\Delta \vdash P_1 \leftrightarrow P_2 \uparrow K$

$\Delta \vdash \alpha \leftrightarrow \alpha \uparrow \Delta(\alpha)$	
$\Delta \vdash \mathsf{unit} \leftrightarrow \mathsf{unit} \uparrow \mathbf{T}$	
$\Delta \vdash C_1' \times C_1'' \leftrightarrow C_2' \times C_2'' \uparrow \mathbf{T}$	if $\Delta \vdash C_1' \Leftrightarrow C_2' : \mathbf{T}$ and $\Delta \vdash C_1'' \Leftrightarrow C_2'' : \mathbf{T}$
$\Delta \vdash C_1' \to C_1'' \leftrightarrow C_2' \to C_2'' \uparrow \mathbf{T}$	if $\Delta \vdash C_1' \Leftrightarrow C_2' : \mathbf{T}$ and $\Delta \vdash C_1'' \Leftrightarrow C_2'' : \mathbf{T}$
$\Delta \vdash \forall\alpha{:}K_1.C_1 \leftrightarrow \forall\alpha{:}K_2.C_2 \uparrow \mathbf{T}$	if $\Delta \vdash K_1 \Leftrightarrow K_2$ and $\Delta, \alpha{:}K_1 \vdash C_1 \Leftrightarrow C_2 : \mathbf{T}$
$\Delta \vdash \exists\alpha{:}K_1.C_1 \leftrightarrow \exists\alpha{:}K_2.C_2 \uparrow \mathbf{T}$	if $\Delta \vdash K_1 \Leftrightarrow K_2$ and $\Delta, \alpha{:}K_1 \vdash C_1 \Leftrightarrow C_2 : \mathbf{T}$
$\Delta \vdash P_1(C_1) \leftrightarrow P_2(C_2) \uparrow K''[C_1/\alpha]$	if $\Delta \vdash P_1 \leftrightarrow P_2 \uparrow \Pi\alpha{:}K'.K''$ and $\Delta \vdash C_1 \Leftrightarrow C_2 : K'$
$\Delta \vdash \pi_1 P_1 \leftrightarrow \pi_1 P_2 \uparrow K'$	if $\Delta \vdash P_1 \leftrightarrow P_2 \uparrow \Sigma\alpha{:}K'.K''$
$\Delta \vdash \pi_2 P_1 \leftrightarrow \pi_2 P_2 \uparrow K''[\pi_1 P_1/\alpha]$	if $\Delta \vdash P_1 \leftrightarrow P_2 \uparrow \Sigma\alpha{:}K'.K''$

Figure 3.9: Equivalence Algorithm for Constructors and Kinds

(namely, C). It is worth noting that this is the only place in the whole equivalence algorithm where the context Δ is actually consulted.

Finally, now that we have reduced C_1 and C_2 to WHNF's P_1 and P_2, we compare the two paths structurally with the judgment $\Delta \vdash P_1 \leftrightarrow P_2 \uparrow K$. In several cases, structural path comparison requires recursive calls to the main equivalence judgment when it encounters subterms, such as function arguments, that are not necessarily paths. The kind K in the path equivalence judgment is the natural kind of P_1. It is used in the function application case to synthesize the kind K' at which the arguments C_1 and C_2 are to be compared.

The proof that this algorithm is sound is fairly straightforward. The proof that it is complete, however, is quite complicated, the chief difficulty being that the algorithm itself is not obviously symmetric or transitive! Specifically, in the pair kind case of the main equivalence judgment, the second recursive call compares $\pi_2 C_1$ and $\pi_2 C_2$ at the kind $K''[\pi_1 C_1/\alpha]$. This does not clearly imply that the two constructors are also algorithmically equivalent at the kind $K''[\pi_1 \underline{C_2}/\alpha]$, which is needed to prove symmetry and transitivity. Similar asymmetries in the kinds pop up in the application and second projection cases of path equivalence.

This problem seems to require one to come up with a variant of the algorithm that is equivalent to it but more obviously symmetric and transitive. Stone and Harper have proposed two such variants. The first, described in their POPL paper [74], overcomes the asymmetries of the original algorithm by working with two equivalent contexts (Δ_1 and Δ_2) and two equivalent kinds (K_1 and K_2) in addition to the two constructors. The idea is to divide the algorithmic judgments into two halves, such that each C_i only ends up "infecting" the kind K_i and context Δ_i on its own half of the judgment. The proof that this algorithm is complete involves a Kripke-style logical relations argument that is fairly straightforward aside from the fact that, like the algorithm, it also deals with two contexts and two kinds.

The clunky nature of the six-place algorithm leads its proof of completeness to be rather verbose. More recently, Stone discovered an alternative algorithm/proof that is, I believe, much easier to follow. While structured like the original algorithm, it takes the form of a normalization procedure for constructors that is both context- and kind-dependent. Two constructors are deemed equivalent if they have the same normal form, so symmetry and transitivity fall out trivially. More interesting is the logical relation used, which has the form $\mathcal{C}$ in $\mathcal{K}$ $[\mathcal{D}]$, where $\mathcal{D}$, $\mathcal{C}$ and $\mathcal{K}$ are *sets* of contexts, constructors and kinds, respectively. The logical relation has the property that all of the constructors in $\mathcal{C}$, when compared at any of the kinds in $\mathcal{K}$ and under any of the contexts in $\mathcal{D}$, have the same normal form. The strengthened induction hypothesis implied by this logical relation results in a completeness proof that is considerably more elegant and readable than the original. It is described in detail in Stone and Harper's forthcoming journal version of their paper [75].

Given soundness and completeness, it is not hard to show that the equivalence algorithm is decidable. One consequence of decidability is that all well-formed constructors have (unique) WHNF's. This is useful particularly when proving decidability of *type* synthesis (see Section 3.2.4 below).

Theorem 3.1.18 (Soundness, Completeness and Decidability of Equivalence Algorithm)

1. If $\Delta \vdash \mathrm{K}_1$ kind and $\Delta \vdash \mathrm{K}_2$ kind, then $\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2$ if and only if $\Delta \vdash \mathrm{K}_1 \Leftrightarrow \mathrm{K}_2$, which is decidable.

2. If $\Delta \vdash \mathrm{C}_1 : \mathrm{K}$ and $\Delta \vdash \mathrm{C}_2 : \mathrm{K}$, then $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}$ if and only if $\Delta \vdash \mathrm{C}_1 \Leftrightarrow \mathrm{C}_2 : \mathrm{K}$, which is decidable.

3. If $\Delta \vdash \mathrm{P}_1 : \mathrm{K}_1$ and $\Delta \vdash \mathrm{P}_2 : \mathrm{K}_2$ and $\Delta \vdash \mathrm{P}_1 \leftrightarrow \mathrm{P}_2 \uparrow \mathrm{K}$, then $\Delta \vdash \mathrm{P}_1 \equiv \mathrm{P}_2 : \mathrm{K}$.

Proposition 3.1.19 (Well-Formed Constructors Have Weak Head Normal Forms)
If $\Delta \vdash \mathrm{C} : \mathrm{K}$, then there exists a unique D such that $\Delta \vdash \mathrm{C} \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{D}$, and moreover $\Delta \vdash \mathrm{C} \equiv \mathrm{D} : \mathrm{K}$.

3.2 Terms

In this section, I present the term layer of my core language. Unlike the constructor and kind languages, this term language is a completely standard explicitly-typed variant of the term language of F_ω.

3.2.1 Syntax

Figure 3.10 gives the syntax of terms and values. I have organized the language so that all introduction forms are values and all values are introduction forms, except for value variables x. A term e is either a value v, an elimination form, or a let-expression. There are two kinds of let-expressions.

Value Variables	$x, y \in$	$\mathit{ValVars}$
Values	$v, w ::=$	$x \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \mathsf{fun}\ x(x_1{:}\mathrm{C}_1){:}\mathrm{C}_2.\, e \mid$
		$\Lambda\alpha{:}\mathrm{K}.e \mid \mathsf{pack}\ [\mathrm{C}, v]\ \mathsf{as}\ \mathrm{D}$
Terms	$e, f ::=$	$v \mid \pi_i v \mid v_1(v_2) \mid v[\mathrm{C}] \mid \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ v\ \mathsf{in}\ (e : \mathrm{C}) \mid$
		$\mathsf{let}\ \alpha = \mathrm{C}\ \mathsf{in}\ e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$
Dynamic Contexts	$\Gamma ::=$	$\emptyset \mid \Gamma, \alpha{:}\mathrm{K} \mid \Gamma, x{:}\mathrm{C}$

Figure 3.10: Syntax of Terms and Values

$$
\begin{array}{lcl}
\langle e_1, e_2\rangle & \stackrel{\text{def}}{=} & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ \langle x_1, x_2\rangle \\
\pi_i e & \stackrel{\text{def}}{=} & \mathsf{let}\ x = e\ \mathsf{in}\ \pi_i x \\
e_1(e_2) & \stackrel{\text{def}}{=} & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ x_1(x_2) \\
e[\mathrm{C}] & \stackrel{\text{def}}{=} & \mathsf{let}\ x = e\ \mathsf{in}\ x[\mathrm{C}] \\
\mathsf{pack}\ [\mathrm{C}, e]\ \mathsf{as}\ \mathrm{D} & \stackrel{\text{def}}{=} & \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{pack}\ [\mathrm{C}, x]\ \mathsf{as}\ \mathrm{D} \\
\mathsf{let}\ [\alpha, y] = \mathsf{unpack}\ e\ \mathsf{in}\ (e' : \mathrm{C}) & \stackrel{\text{def}}{=} & \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{let}\ [\alpha, y] = \mathsf{unpack}\ x\ \mathsf{in}\ (e' : \mathrm{C})
\end{array}
$$

Figure 3.11: Less Restrictive Versions of Term Constructs

The first, $\mathsf{let}\ \alpha = \mathrm{C}\ \mathsf{in}\ e$, binds C to α inside e. This construct is in fact encodable as $(\Lambda\alpha{:}\mathrm{K}.e)[\mathrm{C}]$, where K is the principal kind of C. As this is not a direct syntactic encoding, however, I have included the construct as primitive for convenience. The second let construct, $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, evaluates e_1 to a value v, which is then bound to x inside e_2. Using this construct, we can easily define less restrictive versions of several of the term constructs (shown in Figure 3.11) in which the subterms are permitted to be arbitrary terms and are evaluated in left-to-right order.

Note that the elimination form for existentials, $\mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ v\ \mathsf{in}\ (e : \mathrm{C})$, includes a type annotation C on the result which must be well-formed in the ambient context of the term, *i.e.,* C may not refer to the local type variable α. To eliminate clutter, I will sometimes omit the type annotation C when it is clear from context what it should be.

Finally, Figure 3.10 also defines the syntax of *dynamic contexts*, which extend static contexts with bindings of value variables to their types. There is a straightforward erasure of dynamic contexts into static contexts, defined by the following $\mathsf{Fst}(\cdot)$ function:

$$
\begin{array}{lcl}
\mathsf{Fst}(\emptyset) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{Fst}(\Gamma, \alpha{:}\mathrm{K}) & \stackrel{\text{def}}{=} & \mathsf{Fst}(\Gamma), \alpha{:}\mathrm{K} \\
\mathsf{Fst}(\Gamma, x{:}\mathrm{C}) & \stackrel{\text{def}}{=} & \mathsf{Fst}(\Gamma)
\end{array}
$$

3.2.2 Static Semantics

Figure 3.12 shows the inference rules for the judgments of well-formed terms and dynamic contexts, all of which are straightforward. The only point of note is that the premises of some rules refer to constructor and kind judgments using dynamic contexts Γ in place of static contexts Δ. The meaning of these premises is explained by the following definition:

Well-formed dynamic contexts: $\Gamma \vdash \mathrm{ok}$

$$\frac{}{\emptyset \vdash \mathrm{ok}}\ (52) \qquad \frac{\Gamma \vdash \mathrm{K\ kind}}{\Gamma, \alpha{:}\mathrm{K} \vdash \mathrm{ok}}\ (53) \qquad \frac{\Gamma \vdash \mathrm{C} : \mathbf{T}}{\Gamma, x{:}\mathrm{C} \vdash \mathrm{ok}}\ (54)$$

Well-formed terms: $\Gamma \vdash e : \mathrm{C}$

$$\frac{\Gamma \vdash \mathrm{ok} \quad x{:}\mathrm{C} \in \Gamma}{\Gamma \vdash x : \mathrm{C}}\ (55) \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \langle\rangle : \mathsf{unit}}\ (56)$$

$$\frac{\Gamma \vdash v' : \mathrm{C}' \quad \Gamma \vdash v'' : \mathrm{C}''}{\Gamma \vdash \langle v', v''\rangle : \mathrm{C}' \times \mathrm{C}''}\ (57) \qquad \frac{\Gamma \vdash v : \mathrm{C}_1 \times \mathrm{C}_2 \quad i \in \{1,2\}}{\Gamma \vdash \pi_i v : \mathrm{C}_i}\ (58)$$

$$\frac{\Gamma, x{:}\mathrm{C}' \to \mathrm{C}'', x'{:}\mathrm{C}' \vdash e : \mathrm{C}''}{\Gamma \vdash \mathsf{fun}\ x(x'{:}\mathrm{C}'){:}\mathrm{C}''.\, e : \mathrm{C}' \to \mathrm{C}''}\ (59) \qquad \frac{\Gamma \vdash v : \mathrm{C}' \to \mathrm{C} \quad \Gamma \vdash v' : \mathrm{C}'}{\Gamma \vdash v(v') : \mathrm{C}}\ (60)$$

$$\frac{\Gamma, \alpha{:}\mathrm{K} \vdash e : \mathrm{C}}{\Gamma \vdash \Lambda\alpha{:}\mathrm{K}.e : \forall\alpha{:}\mathrm{K}.\mathrm{C}}\ (61) \qquad \frac{\Gamma \vdash v : \forall\alpha{:}\mathrm{K}.\mathrm{C} \quad \Gamma \vdash \mathrm{D} : \mathrm{K}}{\Gamma \vdash v[\mathrm{D}] : \mathrm{C}[\mathrm{D}/\alpha]}\ (62)$$

$$\frac{\Gamma \vdash \mathrm{D} \equiv \exists\alpha{:}\mathrm{K}.\mathrm{C}' : \mathbf{T} \quad \Gamma \vdash \mathrm{C} : \mathrm{K} \quad \Gamma \vdash v : \mathrm{C}'[\mathrm{C}/\alpha]}{\Gamma \vdash \mathsf{pack}\ [\mathrm{C}, v]\ \mathsf{as}\ \mathrm{D} : \mathrm{D}}\ (63)$$

$$\frac{\Gamma \vdash v : \exists\alpha{:}\mathrm{K}.\mathrm{C}' \quad \Gamma, \alpha{:}\mathrm{K}, x{:}\mathrm{C}' \vdash e : \mathrm{C} \quad \Gamma \vdash \mathrm{C} : \mathbf{T}}{\Gamma \vdash \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ v\ \mathsf{in}\ (e : \mathrm{C}) : \mathrm{C}}\ (64) \qquad \frac{\Gamma \vdash \mathrm{C} : \mathrm{K} \quad \Gamma, \alpha{:}\mathrm{K} \vdash e : \mathrm{D}}{\Gamma \vdash \mathsf{let}\ \alpha = \mathrm{C}\ \mathsf{in}\ e : \mathrm{D}[\mathrm{C}/\alpha]}\ (65)$$

$$\frac{\Gamma \vdash e' : \mathrm{C}' \quad \Gamma, x{:}\mathrm{C}' \vdash e : \mathrm{C}}{\Gamma \vdash \mathsf{let}\ x = e'\ \mathsf{in}\ e : \mathrm{C}}\ (66) \qquad \frac{\Gamma \vdash e : \mathrm{C}' \quad \Gamma \vdash \mathrm{C}' \equiv \mathrm{C} : \mathbf{T}}{\Gamma \vdash e : \mathrm{C}}\ (67)$$

Figure 3.12: Inference Rules for Terms and Dynamic Contexts

Definition 3.2.1 (Static Judgments with Dynamic Contexts)
For any judgment form $\mathcal{J}$ (except "ok") defined in Figures 3.4 and 3.5, let $\Gamma \vdash \mathcal{J}$ be shorthand for the conjunction of $\Gamma \vdash \mathrm{ok}$ and $\mathsf{Fst}(\Gamma) \vdash \mathcal{J}$.

3.2.3 Declarative Properties

It is easy to check that all the propositions stated in Sections 3.1.3 and 3.1.4 involving static contexts may be restated using dynamic contexts instead (*i.e.,* by syntactically replacing all instances of a Δ with a corresponding instance of a Γ), and that all the structural properties (except Substitution) concerning an arbitrary judgment $\mathcal{J}$ apply to the term typing judgment as well. Additionally, we have the following new properties:

Proposition 3.2.2 (Subderivations)
Every proof of $\Gamma_1, x{:}\mathrm{C}, \Gamma_2 \vdash \mathcal{J}$ contains a strict subderivation of $\Gamma_1 \vdash \mathrm{C} : \mathbf{T}$.

Proposition 3.2.3 (Weakening)
If $\Gamma_1, x{:}\mathrm{C}_2, \Gamma_2 \vdash \mathcal{J}$ and $\Gamma_1 \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}$, then $\Gamma_1, x{:}\mathrm{C}_1, \Gamma_2 \vdash \mathcal{J}$.

Proposition 3.2.4 (Validity)
If $\Gamma \vdash e : \mathrm{C}$, then $\Gamma \vdash \mathrm{C} : \mathbf{T}$.

$\boxed{\textbf{Type checking: } \Gamma \vdash e \Leftarrow \mathrm{C}}$

$$\frac{\Gamma \vdash e \Rightarrow \mathrm{C}' \quad \Gamma \vdash \mathrm{C}' \equiv \mathrm{C} : \mathbf{T}}{\Gamma \vdash e \Leftarrow \mathrm{C}}$$

$\boxed{\textbf{Normalized type synthesis: } \Gamma \vdash e \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{C}}$

$$\frac{\Gamma \vdash e \Rightarrow \mathrm{C}' \quad \Gamma \vdash \mathrm{C}' \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{C}}{\Gamma \vdash e \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{C}}$$

$\boxed{\textbf{Type synthesis: } \Gamma \vdash e \Rightarrow \mathrm{C}}$

$$\frac{x{:}\mathrm{C} \in \Gamma}{\Gamma \vdash x \Rightarrow \mathrm{C}} \qquad \frac{}{\Gamma \vdash \langle\rangle \Rightarrow \textsf{unit}} \qquad \frac{\Gamma \vdash v_1 \Rightarrow \mathrm{C}_1 \quad \Gamma \vdash v_2 \Rightarrow \mathrm{C}_2}{\Gamma \vdash \langle v_1, v_2\rangle \Rightarrow \mathrm{C}_1 \times \mathrm{C}_2} \qquad \frac{\Gamma \vdash v \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{C}_1 \times \mathrm{C}_2}{\Gamma \vdash \pi_i v \Rightarrow \mathrm{C}_i}$$

$$\frac{\Gamma \vdash \mathrm{C}' \to \mathrm{C}'' : \mathbf{T} \quad \Gamma, x{:}\mathrm{C}' \to \mathrm{C}'', x'{:}\mathrm{C}' \vdash e \Leftarrow \mathrm{C}''}{\Gamma \vdash \textsf{fun}\ x(x'{:}\mathrm{C}'){:}\mathrm{C}''.\, e \Rightarrow \mathrm{C}' \to \mathrm{C}''} \qquad \frac{\Gamma \vdash v \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{C}' \to \mathrm{C} \quad \Gamma \vdash v' \Leftarrow \mathrm{C}'}{\Gamma \vdash v(v') \Rightarrow \mathrm{C}}$$

$$\frac{\Gamma \vdash \mathrm{K}\ \mathrm{kind} \quad \Gamma, \alpha{:}\mathrm{K} \vdash e \Rightarrow \mathrm{C}}{\Gamma \vdash \Lambda\alpha{:}\mathrm{K}.e \Rightarrow \forall\alpha{:}\mathrm{K}.\mathrm{C}} \qquad \frac{\Gamma \vdash v \stackrel{\mathrm{wh}}{\Longrightarrow} \forall\alpha{:}\mathrm{K}.\mathrm{C} \quad \Gamma \vdash \mathrm{D} : \mathrm{K}}{\Gamma \vdash v[\mathrm{D}] \Rightarrow \mathrm{C}[\mathrm{D}/\alpha]}$$

$$\frac{\Gamma \vdash \mathrm{D} : \mathbf{T} \quad \Gamma \vdash \mathrm{D} \stackrel{\mathrm{wh}}{\Longrightarrow} \exists\alpha{:}\mathrm{K}.\mathrm{C}' \quad \Gamma \vdash \mathrm{C} : \mathrm{K} \quad \Gamma \vdash v \Leftarrow \mathrm{C}'[\mathrm{C}/\alpha]}{\Gamma \vdash \textsf{pack}\ [\mathrm{C}, v]\ \textsf{as}\ \mathrm{D} \Rightarrow \mathrm{D}}$$

$$\frac{\Gamma \vdash v \stackrel{\mathrm{wh}}{\Longrightarrow} \exists\alpha{:}\mathrm{K}.\mathrm{C}' \quad \Gamma, \alpha{:}\mathrm{K}, x{:}\mathrm{C}' \vdash e \Leftarrow \mathrm{C} \quad \Gamma \vdash \mathrm{C} : \mathbf{T}}{\Gamma \vdash \textsf{let}\ [\alpha, x] = \textsf{unpack}\ v\ \textsf{in}\ (e : \mathrm{C}) \Rightarrow \mathrm{C}}$$

$$\frac{\Gamma \vdash \mathrm{C} \Rightarrow \mathrm{K} \quad \Gamma, \alpha{:}\mathrm{K} \vdash e \Rightarrow \mathrm{D}}{\Gamma \vdash \textsf{let}\ \alpha = \mathrm{C}\ \textsf{in}\ e \Rightarrow \mathrm{D}[\mathrm{C}/\alpha]} \qquad \frac{\Gamma \vdash e' \Rightarrow \mathrm{C}' \quad \Gamma, x{:}\mathrm{C}' \vdash e \Rightarrow \mathrm{C}}{\Gamma \vdash \textsf{let}\ x = e'\ \textsf{in}\ e \Rightarrow \mathrm{C}}$$

Figure 3.13: Type Checking and Type Synthesis

3.2.4 Type Checking and Synthesis

Figure 3.13 gives an algorithm for synthesizing the type of a term, which is unique modulo type equivalence. It is completely straightforward. The only point of note is that in several places I make use of weak head normalization in order to reduce a given or synthesized type C into the form of a base type b. Here I state several properties of type synthesis, including that it is sound, complete and deterministic. Decidability is easy to show, as the algorithm is syntax-directed.

Proposition 3.2.5 (Soundness and Other Properties of Type Checking/Synthesis) Assume $\Gamma \vdash \mathrm{ok}$.

1. If $\Gamma \vdash e \Leftarrow \mathrm{C}$ or $\Gamma \vdash e \Rightarrow \mathrm{C}$ or $\Gamma \vdash e \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{C}$, then $\Gamma \vdash e : \mathrm{C}$.

2. Type synthesis is deterministic, *i.e.,* if $\Gamma \vdash e \Rightarrow \mathrm{C}_1$ and $\Gamma \vdash e \Rightarrow \mathrm{C}_2$, then $\mathrm{C}_1 = \mathrm{C}_2$.

3. For $\mathcal{J}$ ranging over any judgment defined in Figure 3.13, if $\Gamma \vdash \mathcal{J}$ and $\Gamma' \supseteq \Gamma$ and $\Gamma' \vdash \mathrm{ok}$, then $\Gamma' \vdash \mathcal{J}$.

$\boxed{\textbf{Small-step semantics: } e \mapsto e'}$

$$\frac{}{\pi_i\langle v_1, v_2\rangle \mapsto v_i} \qquad \frac{v = \mathsf{fun}\ x(x'{:}\mathrm{C}'){:}\mathrm{C}''.\, e}{v(v') \mapsto e[v/x][v'/x']} \qquad \frac{}{(\Lambda\alpha{:}\mathrm{K}.e)[\mathrm{C}] \mapsto e[\mathrm{C}/\alpha]}$$

$$\frac{}{\mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ (\mathsf{pack}\ [\mathrm{C}, v]\ \mathsf{as}\ \mathrm{D})\ \mathsf{in}\ (e : \mathrm{C}') \mapsto e[\mathrm{C}/\alpha][v/x]}$$

$$\frac{}{\mathsf{let}\ \alpha = \mathrm{C}\ \mathsf{in}\ e \mapsto e[\mathrm{C}/\alpha]} \qquad \frac{e_1 \mapsto e_1'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mapsto \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2} \qquad \frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \mapsto e[v/x]}$$

Figure 3.14: Dynamic Semantics of the Core Language

Proposition 3.2.6 (Completeness of Type Checking)
If $\Gamma \vdash e : \mathrm{C}$, then $\Gamma \vdash e \Leftarrow \mathrm{C}$.

3.2.5 Dynamic Semantics and Type Safety

Figure 3.14 gives the dynamic semantics for the core language, in the form of a small-step operational semantics. Again, it is completely standard.

The proof of type safety is done in the usual manner, via preservation and progress theorems, which rely respectively on the substitution and canonical forms lemmas stated below. The proofs of the following properties, except for Substitution, use the type synthesis algorithm of the previous section in order to regularize the structure of term typing derivations.

Lemma 3.2.7 (Substitution for Term Typing Judgment)

1. If $\Gamma_1, \alpha{:}\mathrm{K}, \Gamma_2 \vdash e : \mathrm{D}$ and $\Gamma_1 \vdash \mathrm{C} : \mathrm{K}$, then $\Gamma_1, \Gamma_2[\mathrm{C}/\alpha] \vdash e[\mathrm{C}/\alpha] : \mathrm{D}[\mathrm{C}/\alpha]$.
2. If $\Gamma_1, x{:}\mathrm{C}, \Gamma_2 \vdash e : \mathrm{D}$ and $\Gamma_1, \Gamma_2 \vdash v : \mathrm{C}$, then $\Gamma_1, \Gamma_2 \vdash e[v/x] : \mathrm{D}$.

Lemma 3.2.8 (Canonical Forms)
Suppose $\emptyset \vdash v : \mathrm{C}$. Then:

1. If C is of the form unit, then v is of the form $\langle\rangle$.
2. If C is of the form $\mathrm{C}_1 \times \mathrm{C}_2$, then v is of the form $\langle v_1, v_2\rangle$.
3. If C is of the form $\mathrm{C}_1 \to \mathrm{C}_2$, then v is of the form $\mathsf{fun}\ x(x'{:}\mathrm{C}'){:}\mathrm{C}''.\, e$.
4. If C is of the form $\forall\alpha{:}\mathrm{K}.\mathrm{C}$, then v is of the form $\Lambda\alpha{:}\mathrm{K}'.e$.
5. If C is of the form $\exists\alpha{:}\mathrm{K}.\mathrm{C}$, then v is of the form $\mathsf{pack}\ [\mathrm{C}', v']\ \mathsf{as}\ \mathrm{D}$.

Theorem 3.2.9 (Progress)
If $\emptyset \vdash e : \mathrm{C}$, then either e is a value or there exists a unique e' such that $e \mapsto e'$.

Theorem 3.2.10 (Preservation)
If $\emptyset \vdash e : \mathrm{C}$ and $e \mapsto e'$, then $\emptyset \vdash e' : \mathrm{C}$.

Chapter 4

A Type System for ML Modules: Module Language

In this chapter, I will present the module language of my type system for ML modules, built on top of the core language of Chapter 3. The high-level design of this module language has already been motivated and outlined in Chapter 2. The goal of the present chapter is to explain the formal details and meta-theoretic properties of the language.

In Section 4.1, I will present the language of signatures, which serve as the types of modules. Thanks to the inclusion of singleton kinds in the type structure of the core language, the concept of *translucent* signatures is very easy to account for. I will state and prove a number of properties of signatures, and I will also show how to interpret signatures in terms of core-language kinds and types, via a translation known as "phase-splitting." In Section 4.2, I will present the language of modules itself. While modules do extend the term structure of the core language in order to allow the projection of value components from modules, they do *not* extend the type structure of the core language and thus do not complicate the problem of deciding type equivalence. I will give a sound and complete signature checking algorithm for the module language, and I will also show how to phase-split modules into core-language type constructors and terms, which may then be evaluated according to the core-language dynamic semantics given in Section 3.2.5.

4.1 Signatures

4.1.1 Syntax

The syntax of signatures is given in Figure 4.1. Figure 4.2 illustrates how the constructs in the signature language correspond roughly to the signatures one finds in ML code.[1] Let us look at the signature forms shown in Figure 4.1 one at a time.

The unit signature 1 corresponds to an empty signature with no specifications. The kind signature $[\![\mathrm{K}]\!]$ corresponds to a signature with a single specification of a type constructor with kind K. The type signature $[\![\mathrm{C}]\!]$ models a signature with a single specification of a value component whose type is C.

The pair signature $\Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$ describes a module consisting of a pair of submodules. The signature S_2 of the second submodule may refer to type components of the first submodule, whose

[1]This is only a loose correspondence, meant to provide some intuition. A more precise correspondence will be given in the definition of my new ML dialect in Part III.

Module Variables	$\mathrm{X}, \mathrm{Y} \in$	*ModVars*
Totality Classifiers	$\tau ::=$	$\mathsf{tot} \mid \mathsf{par}$
Signatures	$\mathrm{S}, \mathrm{R} ::=$	$1 \mid [\![\mathrm{K}]\!] \mid [\![\mathrm{C}]\!] \mid \Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \mid \Pi^{\tau}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$
Transparent Signatures	$\mathbb{S}, \mathbb{R} ::=$	$1 \mid [\![\mathbb{K}]\!] \mid [\![\mathrm{C}]\!] \mid \Sigma \mathrm{X}{:}\mathbb{S}_1.\mathbb{S}_2 \mid \Pi^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}_1.\mathbb{S}_2 \mid \Pi^{\mathsf{par}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$

Figure 4.1: Syntax of Signatures

1	$\approx$	`sig end`
$[\![\mathbf{T}]\!]$	$\approx$	`sig type t end`
$[\![\mathbf{T}^n \to \mathbf{T}]\!]$	$\approx$	`sig type ('a`$_1$`,...,'a`$_n$`) t end`
$[\![\mathfrak{S}(\mathrm{C})]\!]$	$\approx$	`sig type t =` C `end`
$[\![\mathrm{C}]\!]$	$\approx$	`sig val x :` C `end`

$\Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$	$\approx$	`sig`
		`structure` X `:` S_1
		`structure` Y `:` S_2
		`end`

$\Pi^{\tau}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$	$\approx$	`functor (`X `:` S_1`) ->` S_2

Figure 4.2: Correspondence With ML Signatures

signature is S_1, through the module variable X. The analogy between $\Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$ and the corresponding ML signature shown in Figure 4.2 is not quite precise: in ML submodules are accessed by name, whereas in this calculus submodules are accessed by position. Hence, while the variable name X is alpha-convertible in $\Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$, changing X to X' in the corresponding ML signature results in an inequivalent signature.

Lastly, the functor signature $\Pi^{\tau}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$ describes a functor with argument signature S_1 and result signature S_2, where S_2 may refer to type components of the argument module through the variable X. The τ is a "totality classifier," which is either tot or par depending on whether the functor is total or partial, respectively. I will sometimes write $\Pi \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$ as shorthand for $\Pi^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$. When $\mathrm{X} \notin \mathrm{FV}(\mathrm{S}_2)$, I will also sometimes use $\mathrm{S}_1 \times \mathrm{S}_2$ and $\mathrm{S}_1 \xrightarrow{\tau} \mathrm{S}_2$ as shorthand for $\Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$ and $\Pi^{\tau}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$, respectively.

Of course, real ML signatures do not have as restricted a form as the signatures in this calculus—they may specify an arbitrary sequence of types, values and submodules. The signature forms in this type system are intended to represent a convenient and concise abstraction of the expressive power of ML signatures, and what they lack in programming flexibility they make up for in simplicity.

There are several points to observe about this signature language. First, in practice, the K in the kind signature $[\![\mathrm{K}]\!]$ will only range over a restricted class of kinds: type constructors at the ML source level are either monomorphic (like `unit`), in which case they have kind $\mathbf{T}$, or polymorphic, (like `list`), in which case they are functions that take some number of arguments n of kind $\mathbf{T}$ and return a type of kind $\mathbf{T}$. ML's "opaque" type specifications thus correspond to kind signatures of the form $[\![\mathbf{T}]\!]$ or $[\![\mathbf{T}^n \to \mathbf{T}]\!]$, which do not reveal any information about the type constructors that

$$
\begin{array}{lcl}
\mathsf{Fst}(1) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{Fst}([\![K]\!]) & \stackrel{\text{def}}{=} & K \\
\mathsf{Fst}([\![C]\!]) & \stackrel{\text{def}}{=} & 1 \\
\mathsf{Fst}(\Sigma X{:}S_1.S_2) & \stackrel{\text{def}}{=} & \Sigma X^{\mathsf{c}}{:}\mathsf{Fst}(S_1).\mathsf{Fst}(S_2) \\
\mathsf{Fst}(\Pi^{\mathsf{tot}} X{:}S_1.S_2) & \stackrel{\text{def}}{=} & \Pi X^{\mathsf{c}}{:}\mathsf{Fst}(S_1).\mathsf{Fst}(S_2) \\
\mathsf{Fst}(\Pi^{\mathsf{par}} X{:}S_1.S_2) & \stackrel{\text{def}}{=} & 1
\end{array}
$$

Figure 4.3: Extracting the Static Part of a Signature

inhabit them (besides their arity n).[2] ML's "transparent" type specifications, in turn, correspond to kind signatures $[\![K]\!]$ where K is a transparent—in particular a singleton—kind.

Second, while the pair and functor signature constructs contain binding sites for module variables, I have failed to provide a way for signatures, constructors or kinds to ever *refer* to those module variables! To remedy this situation, I assume an injection $(\cdot)^{\mathsf{c}}$ from *ModVars* into *ConVars*, that is, for every module variable X, there is a corresponding constructor variable X^{c}, with the property that $X = Y$ if and only if $X^{\mathsf{c}} = Y^{\mathsf{c}}$. In addition, wherever X is bound, X^{c} is implicitly bound as well. The idea is that, if X stands for a module M, then X^{c} is a constructor variable representing the type components of M. Then, instead of projecting types from X directly, which would require extending the type language, we project them from the constructor X^{c}.

The introduction of X^{c} begs the question: if a module variable X has signature S, what is the *kind* of X^{c}? The answer is given by the meta-level function $\mathsf{Fst}(S)$, defined in Figure 4.3. Intuitively, if X has S, then $\mathsf{Fst}(S)$ describes the ways in which type components may be extracted from X (or rather, from X^{c}). If $S = [\![K]\!]$, then X^{c} is the one and only constructor component of X, so it has kind K. If $S = [\![C]\!]$, then X has a single value component and no type components, so X^{c} is equivalent to $\langle\rangle$ and has kind 1. For unit and pair signatures, $\mathsf{Fst}(S)$ is defined in the obvious way.

The definition of $\mathsf{Fst}(S)$ gets slightly tricky when S is a functor signature. When $S = \Pi^{\mathsf{tot}} Y{:}S_1.S_2$, type components may be extracted from X by first applying it to a module with signature S_1 and then projecting types from the result. Since X is a *separably* total functor, however, we know that the type components in X's result can only depend on the *type* components in its argument. Correspondingly, X^{c} is a constructor function from the static part of its argument to the static part of its result and has kind $\Pi Y^{\mathsf{c}}{:}\mathsf{Fst}(S_1).\mathsf{Fst}(S_2)$. On the other hand, when $S = \Pi^{\mathsf{par}} Y{:}S_1.S_2$, there is *no* way to extract type components from X because any application of X will be deemed impure and therefore non-projectible. Consequently, X^{c} is useless—to indicate this, I equate it with the unit constructor $\langle\rangle$ and define $\mathsf{Fst}(\Pi^{\mathsf{par}} Y{:}S_1.S_2)$ to be 1. In the proofs of several properties of the module language, it is convenient to group together signatures S for which $\mathsf{Fst}(S)$ is the unit kind, namely signatures of the form 1, $[\![C]\!]$ or $\Pi^{\mathsf{par}} Y{:}S_1.S_2$. I will refer to these signature forms as *unitary*.

Finally, analogous to the notion of higher-order singleton kinds, Figure 4.4 defines a notion of singleton signatures.[3] Intuitively, the singleton signature $\mathfrak{S}_S(C)$ describes precisely those modules of signature S whose static parts (of kind $\mathsf{Fst}(S)$) are equivalent to C. Given this intuition, the formal definition itself is fairly straightforward. Singleton signatures will come in very handy in

[2] $\mathbf{T}^n$ here stands for $\underbrace{\mathbf{T} \times \cdots \times \mathbf{T}}_{n \text{ times}}$.

[3] Similarly to higher-order singleton kinds, $\mathfrak{S}_S(C)$ is defined inductively on the *size* of the signature S, using the size metric of Definition 4.1.2.

$$
\begin{array}{lcl}
\mathfrak{S}_{1}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & 1 \\
\mathfrak{S}_{[\![\mathrm{K}]\!]}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & [\![\mathfrak{S}_{\mathrm{K}}(\mathrm{C})]\!] \\
\mathfrak{S}_{[\![\mathrm{D}]\!]}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & [\![\mathrm{D}]\!] \\
\mathfrak{S}_{\Sigma \mathrm{X}:\mathrm{S}_1.\mathrm{S}_2}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & \mathfrak{S}_{\mathrm{S}_1}(\pi_1\mathrm{C}) \times \mathfrak{S}_{\mathrm{S}_2[\pi_1\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\pi_2\mathrm{C}) \\
\mathfrak{S}_{\Pi^{\mathsf{tot}}\mathrm{X}:\mathrm{S}_1.\mathrm{S}_2}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & \Pi^{\mathsf{tot}}\mathrm{X}:\mathrm{S}_1.\mathfrak{S}_{\mathrm{S}_2}(\mathrm{C}(\mathrm{X}^{\mathsf{c}})) \\
\mathfrak{S}_{\Pi^{\mathsf{par}}\mathrm{X}:\mathrm{S}_1.\mathrm{S}_2}(\mathrm{C}) & \stackrel{\mathrm{def}}{=} & \Pi^{\mathsf{par}}\mathrm{X}:\mathrm{S}_1.\mathrm{S}_2
\end{array}
$$

Figure 4.4: Singleton Signatures

formalizing the idea of selfification at the level of modules in much the same way that higher-order singleton kinds formalize selfification at the level of type constructors.

Singleton signatures are a special case of a more general subclass of *transparent* signatures, written $\mathbb{S}$, whose syntax is defined in Figure 4.1. Conceptually, a transparent signature is a signature in which the type components (if there are any) are fully specified, *i.e.,* have transparent kinds. More formally, S is a transparent signature if and only if $\mathsf{Fst}(\mathrm{S})$ is a transparent kind. One consequence of this definition is that all unitary signatures are considered transparent—in particular, a partial functor signature is considered transparent, even if its result signature is not. This makes sense: since no type components may be extracted from a partial functor, it is indeed the case that all zero of them are transparently specified. The ability to syntactically distinguish transparent signatures is useful both in the typing judgment for modules (presented in Section 4.2.3) and in the meta-theory of the module language.

4.1.2 Static Semantics

Figure 4.5 shows the inference rules for the judgments of well-formed signatures, signature equivalence and signature subtyping. The important thing to note about all three judgments is that they employ a static context Δ, not a dynamic context Γ. Thus, in the cases of Σ and Π signatures, instead of binding the module variable X in the context[4] with signature S_1, we bind X^{c} in the context with $\mathsf{Fst}(\mathrm{S}_1)$. As explained in the previous section, this is feasible because S_2 only needs to refer to the *type* components of X, which are represented by X^{c}. I have defined the judgments in this way so as to emphasize the point that signatures are purely static entities, which may only depend on other static entities such as constructors and kinds, not on dynamic values.

The well-formedness and equivalence judgments are completely straightforward. The subtyping judgment, however, requires a bit of explanation. The point of signature subtyping in this calculus is to allow the identities of a module's type components to be forgotten when coercing the module from a more transparent signature to a more opaque signature. This is not as liberal as signature matching in ML, which additionally permits both the dropping and reordering of components and the specialization of polymorphic value components. These other features of ML's signature matching are important, but they can be supported by generating explicit coercions between signatures (see Chapter 9 for details).

The subtyping defined in Figure 4.5 allows for the forgetting of a module's type components via Rule 79, which uses kind subtyping to implement the forgetfulness. Subtyping is defined at pair

[4]The ability to bind module variables in the context will be introduced in Section 4.2.1.

Well-formed signatures: $\Delta \vdash \mathrm{S}\ \mathrm{sig}$

$$\frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash 1\ \mathrm{sig}}\ (68) \qquad \frac{\Delta \vdash \mathrm{K}\ \mathrm{kind}}{\Delta \vdash [\![\mathrm{K}]\!]\ \mathrm{sig}}\ (69) \qquad \frac{\Delta \vdash \mathrm{C} : \mathbf{T}}{\Delta \vdash [\![\mathrm{C}]\!]\ \mathrm{sig}}\ (70)$$

$$\frac{\Delta \vdash \mathrm{S}'\ \mathrm{sig} \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}') \vdash \mathrm{S}''\ \mathrm{sig}}{\Delta \vdash \Sigma \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''\ \mathrm{sig}}\ (71) \qquad \frac{\Delta \vdash \mathrm{S}'\ \mathrm{sig} \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}') \vdash \mathrm{S}''\ \mathrm{sig}}{\Delta \vdash \Pi^{\tau} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''\ \mathrm{sig}}\ (72)$$

Signature equivalence: $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$

$$\frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash 1 \equiv 1}\ (73) \qquad \frac{\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2}{\Delta \vdash [\![\mathrm{K}_1]\!] \equiv [\![\mathrm{K}_2]\!]}\ (74) \qquad \frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Delta \vdash [\![\mathrm{C}_1]\!] \equiv [\![\mathrm{C}_2]\!]}\ (75)$$

$$\frac{\Delta \vdash \mathrm{S}_1' \equiv \mathrm{S}_2' \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1') \vdash \mathrm{S}_1'' \equiv \mathrm{S}_2''}{\Delta \vdash \Sigma \mathrm{X}{:}\mathrm{S}_1'.\mathrm{S}_1'' \equiv \Sigma \mathrm{X}{:}\mathrm{S}_2'.\mathrm{S}_2''}\ (76) \qquad \frac{\Delta \vdash \mathrm{S}_2' \equiv \mathrm{S}_1' \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_2') \vdash \mathrm{S}_1'' \equiv \mathrm{S}_2''}{\Delta \vdash \Pi^{\tau} \mathrm{X}{:}\mathrm{S}_1'.\mathrm{S}_1'' \equiv \Pi^{\tau} \mathrm{X}{:}\mathrm{S}_2'.\mathrm{S}_2''}\ (77)$$

Signature subtyping: $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$

$$\frac{\Delta \vdash \mathrm{ok}}{\Delta \vdash 1 \leq 1}\ (78) \qquad \frac{\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2}{\Delta \vdash [\![\mathrm{K}_1]\!] \leq [\![\mathrm{K}_2]\!]}\ (79) \qquad \frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Delta \vdash [\![\mathrm{C}_1]\!] \leq [\![\mathrm{C}_2]\!]}\ (80)$$

$$\frac{\Delta \vdash \mathrm{S}_1' \leq \mathrm{S}_2' \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1') \vdash \mathrm{S}_1'' \leq \mathrm{S}_2'' \quad \Delta \vdash \Sigma \mathrm{X}{:}\mathrm{S}_2'.\mathrm{S}_2''\ \mathrm{sig}}{\Delta \vdash \Sigma \mathrm{X}{:}\mathrm{S}_1'.\mathrm{S}_1'' \leq \Sigma \mathrm{X}{:}\mathrm{S}_2'.\mathrm{S}_2''}\ (81)$$

$$\frac{\Delta \vdash \mathrm{S}_2' \equiv \mathrm{S}_1' \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_2') \vdash \mathrm{S}_1'' \leq \mathrm{S}_2''}{\Delta \vdash \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}_1'.\mathrm{S}_1'' \leq \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}_2'.\mathrm{S}_2''}\ (82) \qquad \frac{\Delta \vdash \mathrm{S}_2' \equiv \mathrm{S}_1' \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_2') \vdash \mathrm{S}_1'' \equiv \mathrm{S}_2''}{\Delta \vdash \Pi^{\mathsf{par}} \mathrm{X}{:}\mathrm{S}_1'.\mathrm{S}_1'' \leq \Pi^{\mathsf{par}} \mathrm{X}{:}\mathrm{S}_2'.\mathrm{S}_2''}\ (83)$$

Figure 4.5: Inference Rules for Signatures

signatures in the standard covariant manner. For functor signatures, though, subtyping is very restrictive: total functor signatures are not considered subtypes of partial functor signatures, and subtyping for both kinds of signatures is invariant, not contravariant, in the argument. In fact, for partial signatures, subtyping is even invariant in the result. I will explain the specific technical reasons for these restrictions in Section 4.1.4, but essentially they are due to the lack of subtyping—at the level of types, not kinds—in the core language. Nevertheless, as with the other features of ML's signature matching, it is possible to define a more standard co- and contra-variant formulation of functor subtyping by means of explicit module coercions, and I will do so in Chapter 9.

4.1.3 Declarative Properties

Unlike the core language of Chapter 3, the module language of this chapter is presented here in a new formulation for which the meta-theory has not previously been written down. I will therefore give proofs of any theorems that are not entirely straightforward. Fortunately, there are not many.

First, it is easy to check that the basic structural properties stated in Section 3.1.3 (Propositions 3.1.1, 3.1.2, 3.1.4, and 3.1.5) all hold for the new signature judgments defined above. The Substitution property relies on the fact that substitution commutes with $\mathsf{Fst}(\mathrm{S})$ and $\mathfrak{s}_{\mathrm{S}}(\mathrm{C})$, as stated by the following proposition:

Proposition 4.1.1 (Substitution Commutes With $\mathsf{Fst}(\mathrm{S})$ and $\mathfrak{s}_{\mathrm{S}}(\mathrm{C})$)

1. $\gamma(\mathsf{Fst}(\mathrm{S})) = \mathsf{Fst}(\gamma\mathrm{S})$.

2. $\gamma(\mathfrak{s}_{\mathrm{S}}(\mathrm{C})) = \mathfrak{s}_{\gamma\mathrm{S}}(\gamma\mathrm{C})$.

With the structural properties in hand, we now state some basic facts about $\mathsf{Fst}(\mathrm{S})$ and $\mathfrak{s}_{\mathrm{S}}(\mathrm{C})$, namely: that they commute with each other, that they take well-formed arguments to well-formed results, and that Fst preserves the equivalence/subtyping relationships of its arguments.[5] It is useful to prove these facts before anything else, since $\mathsf{Fst}(\cdot)$ at least is ubiquitous in the signature judgments. Luckily, this is completely straightforward. The proofs of these and other properties of signatures are by induction on the size of signatures, as defined by the following metric:

Definition 4.1.2 (Sizes of Signatures)
Let the *size* of a signature S, written size(S), be defined inductively as follows:

$$\begin{array}{lcl}
\mathrm{size}(1) & \stackrel{\mathrm{def}}{=} & 1 \\
\mathrm{size}([\![\mathrm{K}]\!]) & \stackrel{\mathrm{def}}{=} & 2 \\
\mathrm{size}([\![\mathrm{C}]\!]) & \stackrel{\mathrm{def}}{=} & 2 \\
\mathrm{size}(\Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2) & \stackrel{\mathrm{def}}{=} & 1 + \mathrm{size}(\mathrm{S}_1) + \mathrm{size}(\mathrm{S}_2) \\
\mathrm{size}(\Pi^{\tau} \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2) & \stackrel{\mathrm{def}}{=} & 1 + \mathrm{size}(\mathrm{S}_1) + \mathrm{size}(\mathrm{S}_2)
\end{array}$$

Note that $\mathrm{size}(\mathfrak{s}_{\mathrm{S}}(\mathrm{C})) = \mathrm{size}(\mathrm{S})$ for all S and C.

Proposition 4.1.3 (Facts About $\mathsf{Fst}(\mathrm{S})$ and $\mathfrak{s}_{\mathrm{S}}(\mathrm{C})$)

1. If $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathrm{S})$, then $\Delta \vdash \mathsf{Fst}(\mathfrak{s}_{\mathrm{S}}(\mathrm{C})) \equiv \mathfrak{s}_{\mathsf{Fst}(\mathrm{S})}(\mathrm{C})$.

2. If $\Delta \vdash \mathrm{S}\ \mathrm{sig}$, then $\Delta \vdash \mathsf{Fst}(\mathrm{S})\ \mathrm{kind}$.

3. If $\Delta \vdash \mathrm{S}\ \mathrm{sig}$ and $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathrm{S})$, then $\Delta \vdash \mathfrak{s}_{\mathrm{S}}(\mathrm{C})\ \mathrm{sig}$.

4. If $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$, then $\Delta \vdash \mathsf{Fst}(\mathrm{S}_1) \equiv \mathsf{Fst}(\mathrm{S}_2)$.

5. If $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$, then $\Delta \vdash \mathsf{Fst}(\mathrm{S}_1) \leq \mathsf{Fst}(\mathrm{S}_2)$.

Proof: By induction on the size of the given signature(s). ■

The following properties of signatures, which are all analogous to properties of kinds, also admit completely straightforward proofs by induction on the sizes of the given signatures. Those familiar with the Stone-Harper meta-theory may be surprised that the proofs of validity and functionality for signatures are completely straightforward, since at the kind level they are not. At the kind level, validity and functionality are intertwined: kind-level validity depends on constructor-level validity, which in turn depends on kind-level functionality, and the proof of functionality itself depends on validity. This cycle can be broken, but it requires some work (the Stone-Harper technical report [73] and Stone's thesis [72] show two different ways to do it). Such a cycle does not occur, however, at the signature level: signature-level validity depends only on constructor-level validity and kind-level validity, both of which have already been proven.

Proposition 4.1.4 (Reflexivity)
If $\Delta \vdash \mathrm{S}\ \mathrm{sig}$, then $\Delta \vdash \mathrm{S} \equiv \mathrm{S}$ and $\Delta \vdash \mathrm{S} \leq \mathrm{S}$.

---

[5] The singleton signature macro also preserves the equivalence/subtyping relationships of its arguments, but it is easier to prove this later (see Proposition 4.1.9).

Proposition 4.1.5 (Validity)
If $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$ or $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$, then $\Delta \vdash \mathrm{S}_1\ \mathrm{sig}$ and $\Delta \vdash \mathrm{S}_2\ \mathrm{sig}$.

Proposition 4.1.6 (Symmetry and Transitivity of Signature Equivalence)

1. If $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$, then $\Delta \vdash \mathrm{S}_2 \equiv \mathrm{S}_1$.
2. If $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$ and $\Delta \vdash \mathrm{S}_2 \equiv \mathrm{S}_3$, then $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_3$.

Proposition 4.1.7 (Antisymmetry and Transitivity of Signature Subtyping)

1. $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$ if and only if $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$ and $\Delta \vdash \mathrm{S}_2 \leq \mathrm{S}_1$.
2. If $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$ and $\Delta \vdash \mathrm{S}_2 \leq \mathrm{S}_3$, then $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_3$.

Proposition 4.1.8 (Functionality)
Suppose $\Delta' \vdash \gamma_1 \equiv \gamma_2 : \Delta$.

1. If $\Delta \vdash \mathrm{S}\ \mathrm{sig}$, then $\Delta' \vdash \gamma_1\mathrm{S} \equiv \gamma_2\mathrm{S}$.
2. If $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$, then $\Delta' \vdash \gamma_1\mathrm{S}_1 \equiv \gamma_2\mathrm{S}_2$.
3. If $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$, then $\Delta' \vdash \gamma_1\mathrm{S}_1 \leq \gamma_2\mathrm{S}_2$.

The proofs of the following properties of singleton and transparent signatures are completely analogous to the proofs of Propositions 3.1.12 and 3.1.14 given in Section 3.1.5. The only substantively new cases are the type-specification signature $[\![\mathrm{K}]\!]$, for which the proof follows directly from the propositions just named, and the unitary signatures, for which the proof is trivial because $\mathfrak{s}_{\mathrm{S}}(\mathrm{C}) = \mathrm{S}$ when S is unitary. The proofs implicitly make use of the fact that Fst commutes with the singleton signature macro (Proposition 4.1.3 above).

Proposition 4.1.9 (Singleton and Transparent Signature Rules)

1. If $\Delta \vdash \mathrm{S}\ \mathrm{sig}$ and $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathrm{S})$, then $\Delta \vdash \mathfrak{s}_{\mathrm{S}}(\mathrm{C}) \leq \mathrm{S}$.
2. If $\Delta \vdash \mathbb{S}\ \mathrm{sig}$ and $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathbb{S})$, then $\Delta \vdash \mathfrak{s}_{\mathbb{S}}(\mathrm{C}) \equiv \mathbb{S}$.
3. If $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$ and $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathsf{Fst}(\mathrm{S}_1)$, then $\Delta \vdash \mathfrak{s}_{\mathrm{S}_1}(\mathrm{C}_1) \leq \mathfrak{s}_{\mathrm{S}_2}(\mathrm{C}_2)$.
4. If $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$ and $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathsf{Fst}(\mathrm{S}_1)$, then $\Delta \vdash \mathfrak{s}_{\mathrm{S}_1}(\mathrm{C}_1) \equiv \mathfrak{s}_{\mathrm{S}_2}(\mathrm{C}_2)$.
5. If $\Delta \vdash \mathrm{S}_1 \leq \mathbb{S}_2$, then S_1 is transparent.
6. $\mathfrak{s}_{\mathrm{S}}(\mathrm{C})$ is transparent.
7. S is transparent if and only if $\mathsf{Fst}(\mathrm{S})$ is transparent.

Proof: By induction on the size of the given signature(s).

1.
   - Case: S is unitary. Trivial, by reflexivity.
   - Case: $\mathrm{S} = [\![\mathrm{K}]\!]$. By Proposition 3.1.12, $\Delta \vdash \mathfrak{s}_{\mathrm{K}}(\mathrm{C}) \leq \mathrm{K}$, so $\Delta \vdash [\![\mathfrak{s}_{\mathrm{K}}(\mathrm{C})]\!] \leq [\![\mathrm{K}]\!]$.
   - Case: $\mathrm{S} = \Sigma \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''$.
     (a) By inversion, $\Delta \vdash \mathrm{S}'\ \mathrm{sig}$ and $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}') \vdash \mathrm{S}''\ \mathrm{sig}$.
     (b) Since $\Delta \vdash \pi_1\mathrm{C} : \mathsf{Fst}(\mathrm{S}')$, by induction $\Delta \vdash \mathfrak{s}_{\mathrm{S}'}(\pi_1\mathrm{C}) \leq \mathrm{S}'$,
     (c) and by Substitution, $\Delta \vdash \mathrm{S}''[\pi_1\mathrm{C}/\mathrm{X}^{\mathsf{c}}]\ \mathrm{sig}$.

(d) Since $\Delta \vdash \pi_2 C : \mathsf{Fst}(S'')[\pi_1 C/X^c]$, by induction $\Delta \vdash \mathfrak{s}_{S''[\pi_1 C/X^c]}(\pi_2 C) \leq S''[\pi_1 C/X^c]$.
(e) By Proposition 3.1.12, $\Delta, X^c{:}\mathfrak{s}_{\mathsf{Fst}(S')}(\pi_1 C) \vdash \pi_1 C \equiv X^c : \mathfrak{s}_{\mathsf{Fst}(S')}(\pi_1 C)$.
(f) By Functionality, $\Delta, X^c{:}\mathsf{Fst}(\mathfrak{s}_{S'}(\pi_1 C)) \vdash S''[\pi_1 C/X^c] \leq S''$.
(g) Thus, $\Delta \vdash \mathfrak{s}_{S'}(\pi_1 C) \times \mathfrak{s}_{S''[\pi_1 C/X^c]}(\pi_2 C) \leq \Sigma X{:}S'.S''$.

- Case: $S = \Pi X{:}S'.S''$.
  (a) Since $\Delta, X^c{:}\mathsf{Fst}(S') \vdash S''$ sig, and $\Delta, X^c{:}\mathsf{Fst}(S') \vdash C(X^c) : \mathsf{Fst}(S'')$,
  (b) by induction $\Delta, X^c{:}\mathsf{Fst}(S') \vdash \mathfrak{s}_{S''}(C(X^c)) \leq S''$.
  (c) Thus, $\Delta \vdash \Pi X{:}S'.\mathfrak{s}_{S''}(C(X^c)) \leq \Pi X{:}S'.S''$.

2.
- Case: $\mathbb{S}$ is unitary. Trivial, by reflexivity.
- Case: $\mathbb{S} = [\![\mathbb{K}]\!]$. By Proposition 3.1.14, $\Delta \vdash \mathfrak{s}_{\mathbb{K}}(C) \equiv \mathbb{K}$, so $\Delta \vdash [\![\mathfrak{s}_{\mathbb{K}}(C)]\!] \equiv [\![\mathbb{K}]\!]$.
- Case: $\mathbb{S} = \Sigma X{:}\mathbb{S}'.\mathbb{S}''$.
  (a) By inversion, $\Delta \vdash \mathbb{S}'$ sig and $\Delta, X^c{:}\mathsf{Fst}(\mathbb{S}') \vdash \mathbb{S}''$ sig.
  (b) Since $\Delta \vdash \pi_1 C : \mathsf{Fst}(\mathbb{S}')$, by induction $\Delta \vdash \mathfrak{s}_{\mathbb{S}'}(\pi_1 C) \equiv \mathbb{S}'$,
  (c) and by Substitution, $\Delta \vdash \mathbb{S}''[\pi_1 C/X^c]$ sig.
  (d) Since $\Delta \vdash \pi_2 C : \mathsf{Fst}(\mathbb{S}'')[\pi_1 C/X^c]$, by induction $\Delta \vdash \mathfrak{s}_{\mathbb{S}''[\pi_1 C/X^c]}(\pi_2 C) \equiv \mathbb{S}''[\pi_1 C/X^c]$.
  (e) By Proposition 3.1.12, $\Delta, X^c{:}\mathfrak{s}_{\mathsf{Fst}(\mathbb{S}')}(\pi_1 C) \vdash \pi_1 C \equiv X^c : \mathfrak{s}_{\mathsf{Fst}(\mathbb{S}')}(\pi_1 C)$.
  (f) By Functionality, $\Delta, X^c{:}\mathsf{Fst}(\mathfrak{s}_{\mathbb{S}'}(\pi_1 C)) \vdash \mathbb{S}''[\pi_1 C/X^c] \equiv \mathbb{S}''$.
  (g) Thus, $\Delta \vdash \mathfrak{s}_{\mathbb{S}'}(\pi_1 C) \times \mathfrak{s}_{\mathbb{S}''[\pi_1 C/X^c]}(\pi_2 C) \equiv \Sigma X{:}\mathbb{S}'.\mathbb{S}''$.
- Case: $\mathbb{S} = \Pi X{:}S'.\mathbb{S}''$.
  (a) Since $\Delta, X^c{:}\mathsf{Fst}(S') \vdash \mathbb{S}''$ sig, and $\Delta, X^c{:}\mathsf{Fst}(S') \vdash C(X^c) : \mathsf{Fst}(\mathbb{S}'')$,
  (b) by induction $\Delta, X^c{:}\mathsf{Fst}(S') \vdash \mathfrak{s}_{\mathbb{S}''}(C(X^c)) \equiv \mathbb{S}''$.
  (c) Thus, $\Delta \vdash \Pi X{:}S'.\mathfrak{s}_{\mathbb{S}''}(C(X^c)) \equiv \Pi X{:}S'.\mathbb{S}''$.

3.
- Case: S_1 and S_2 are unitary. Trivial, by assumption.
- Case: $S_i = [\![K_i]\!]$. By Proposition 3.1.12, $\Delta \vdash \mathfrak{s}_{K_1}(C_1) \leq \mathfrak{s}_{K_2}(C_2)$, so $\Delta \vdash [\![\mathfrak{s}_{K_1}(C_1)]\!] \leq [\![\mathfrak{s}_{K_2}(C_2)]\!]$.
- Case: $S_i = \Sigma X{:}S_i'.S_i''$.
  (a) By inversion, $\Delta \vdash S_1' \leq S_2'$ and $\Delta, X^c{:}\mathsf{Fst}(S_1') \vdash S_1'' \leq S_2''$.
  (b) Since $\Delta \vdash \pi_1 C_1 \equiv \pi_1 C_2 : \mathsf{Fst}(S_1')$, by induction $\Delta \vdash \mathfrak{s}_{S_1'}(\pi_1 C_1) \leq \mathfrak{s}_{S_2'}(\pi_1 C_2)$,
  (c) and by Functionality, $\Delta \vdash S_1''[\pi_1 C_1/X^c] \leq S_2''[\pi_1 C_2/X^c]$.
  (d) Since $\Delta \vdash \pi_2 C_1 \equiv \pi_2 C_2 : \mathsf{Fst}(S_1'')[\pi_1 C_1/X^c]$,
  (e) by induction $\Delta \vdash \mathfrak{s}_{S_1''[\pi_1 C_1/X^c]}(\pi_2 C_1) \leq \mathfrak{s}_{S_2''[\pi_1 C_2/X^c]}(\pi_2 C_2)$.
  (f) Thus, $\Delta \vdash \mathfrak{s}_{S_1'}(\pi_1 C_1) \times \mathfrak{s}_{S_1''[\pi_1 C_1/X^c]}(\pi_2 C_1) \leq \mathfrak{s}_{S_2'}(\pi_1 C_2) \times \mathfrak{s}_{S_2''[\pi_1 C_2/X^c]}(\pi_2 C_2)$.

4. By Antisymmetry, Part 3 of this proposition, and Part 4 of Proposition 4.1.3.

5–7. Straightforward.

■

$\boxed{\textbf{Signature phase-splitting: } \mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}]\!]}$

$$\frac{}{1 \rightrightarrows [\![\alpha{:}1.\mathsf{unit}]\!]} \qquad \frac{}{[\![\mathrm{K}]\!] \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathsf{unit}]\!]} \qquad \frac{}{[\![\mathrm{C}]\!] \rightrightarrows [\![\alpha{:}1.\mathrm{C}]\!]}$$

$$\frac{\mathrm{S}_1 \rightrightarrows [\![\mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1.\mathrm{C}_1]\!] \quad \mathrm{S}_2 \rightrightarrows [\![\alpha_2{:}\mathrm{K}_2.\mathrm{C}_2]\!]}{\Sigma\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\![\alpha{:}(\Sigma\mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1.\mathrm{K}_2).\mathrm{C}_1[\pi_1\alpha/\mathrm{X}^{\mathsf{c}}] \times \mathrm{C}_2[\pi_1\alpha/\mathrm{X}^{\mathsf{c}}][\pi_2\alpha/\alpha_2]]\!]}$$

$$\frac{\mathrm{S}_1 \rightrightarrows [\![\mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1.\mathrm{C}_1]\!] \quad \mathrm{S}_2 \rightrightarrows [\![\alpha_2{:}\mathrm{K}_2.\mathrm{C}_2]\!]}{\Pi^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\![\alpha{:}(\Pi\mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1.\mathrm{K}_2).\forall\mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1.\mathrm{C}_1 \to \mathrm{C}_2[\alpha(\mathrm{X}^{\mathsf{c}})/\alpha_2]]\!]}$$

$$\frac{\mathrm{S}_1 \rightrightarrows [\![\mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1.\mathrm{C}_1]\!] \quad \mathrm{S}_2 \rightrightarrows [\![\alpha_2{:}\mathrm{K}_2.\mathrm{C}_2]\!]}{\Pi^{\mathsf{par}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\![\alpha{:}1.\forall\mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1.\mathrm{C}_1 \to \exists\alpha_2{:}\mathrm{K}_2.\mathrm{C}_2]\!]}$$

$$\langle\!|\mathrm{S}|\!\rangle \stackrel{\text{def}}{=} \exists\alpha{:}\mathrm{K}.\mathrm{C}, \text{ where } \mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}]\!]$$

Figure 4.6: Signature Phase-Splitting and Definition of Package Type

4.1.4 Signature Phase-Splitting

In this section, I define a so-called "phase-splitting" translation from the signature language into the type structure of the core language, and show that it is sound and preserves equivalence and subtyping relations. The phase-splitting translation shown in Figure 4.6 takes the form of a judgment $\mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}]\!]$, where S is the signature being translated, K represents the "static" part of S (*i.e.,* the specifications of S's type components) and C represents the "dynamic" part of the signature (*i.e.,* the specifications of S's value components), which may refer to the type components through the constructor variable α. Note that the static part K is precisely $\mathsf{Fst}(\mathrm{S})$. Figure 4.6 also defines the "package type" $\langle\!|\mathrm{S}|\!\rangle$ to be the existential type formed from packaging together the static and dynamic parts of S. Package types will be used in the next section to classify modules that have been pack'ed as core-language terms.

The phase-splitting translations of unit, kind and type signatures are all self-explanatory. The translations of the remaining signatures[6] are easiest to think of in terms of how they guide the phase-splitting of modules, which will be formalized in Section 4.2.7. For a module M of pair signature, the translation joins the static parts of M's first and second components as a pair of constructors, and joins the dynamic parts as a pair of terms. For a total functor F of signature $\Pi^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$, the fact that "total" here means "*separably* total" tells us that the static part of F's result only depends on the static part of its argument, so the static part of F itself is a constructor function of kind $\Pi\mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1).\mathsf{Fst}(\mathrm{S}_2)$. The dynamic part of F's result, however, may depend on both the static and dynamic parts of its argument, so F's dynamic part is a polymorphic function taking both a type and a value argument. When F is a partial functor, the static part of its result cannot be hoisted out, because it may depend on effectful operations that produce different results at each application of F. Correspondingly, F is translated as a polymorphic function returning an existential package whose static part is unknown. Note that the result type of this function is precisely $\langle\!|\mathrm{S}_2|\!\rangle$.

The following proposition states a number of properties of signature phase-splitting, namely

[6]The rules for pair and total functor signatures follow exactly the non-standard signature equivalence rules employed by Harper, Mitchell and Moggi in their phase-distinction calculus [30].

that: (1) it commutes with substitution, (2) the static part of S is precisely $\mathsf{Fst}(S)$, (3) well-formed signatures phase-split to well-formed results, (4) equivalent signatures phase-split to equivalent results, and (5) if S_1 is a subtype of S_2, then the dynamic parts of the signatures are equivalent, but $\mathsf{Fst}(S_1)$ is only a subkind of $\mathsf{Fst}(S_2)$.

Part 5, which I need in order to prove soundness of module phase-splitting in Section 4.2.7, is what motivates my restrictive definition of functor subtyping. Specifically, for any functor signature $S = \Pi^\tau X{:}S_1.S_2$, the static part of S_1 leaks into the dynamic part of S, so the dynamic parts of two functor signatures will only be equivalent if their argument signatures have equivalent static parts. This condition will not be met if the argument signatures are merely in a contravariant subtyping relationship. In the case that S is partial, the static part of S_2 leaks into the dynamic part of S as well, so the dynamic parts of two partial functor signatures will only be equivalent if their result signatures have equivalent static parts. This condition will not be met if the result signatures are merely in a covariant subtyping relationship.

It is worth noting that if the core language supported subtyping in addition to subkinding, and if Part 5 were weakened to only require that C_1 be a *subtype* of C_2, then the restrictions I have imposed on functor subtyping could be avoided. For simplicity, however, I have chosen not to introduce subtyping into the core language.

Proposition 4.1.10 (Soundness and Other Properties of Signature Phase-Splitting)

1. If $S \rightrightarrows [\![\alpha{:}K.C]\!]$, then $\gamma S \rightrightarrows [\![\alpha{:}\gamma K.\gamma C]\!]$.

2. If $S \rightrightarrows [\![\alpha{:}K.C]\!]$, then $K = \mathsf{Fst}(S)$.

3. If $\Delta \vdash S$ sig, then $\Delta \vdash \langle\!|S|\!\rangle : \mathbf{T}$.

4. If $\Delta \vdash S_1 \equiv S_2$, then $\Delta \vdash \langle\!|S_1|\!\rangle \equiv \langle\!|S_2|\!\rangle : \mathbf{T}$.

5. If $\Delta \vdash S_1 \leq S_2$ and $S_1 \rightrightarrows [\![\alpha{:}K_1.C_1]\!]$ and $S_2 \rightrightarrows [\![\alpha{:}K_2.C_2]\!]$, then $\Delta \vdash K_1 \leq K_2$ and $\Delta, \alpha{:}K_1 \vdash C_1 \equiv C_2 : \mathbf{T}$.

Proof: By straightforward induction on the size of the given signature(s). ∎

4.2 Modules

4.2.1 Syntax

The syntax of modules is given in Figure 4.7, along with extensions to the syntax of terms. Dynamic contexts are also extended with a new binding form that assigns signatures to module variables. Figure 4.8 illustrates how some of the module constructs correspond to module expressions one finds in dialects of ML.

The unit module containing no bindings is written $\langle\rangle$. The constructor module [C] contains a single binding of a constructor component defined as C. The term module $[e]$ contains a single binding of a value component defined by evaluating the term e. The new term-level construct $\mathsf{Term}(M)$ allows one to extract the single value component from a module M of signature $[\![C]\!]$.

The pair module $\langle X = M_1, M_2\rangle$ consists of a pair of modules, wherein the second module M_2 may refer to the first (or rather, to the result of evaluating the first) by the variable X. As with pair signatures, the ML analogue for $\langle X = M_1, M_2\rangle$ shown in Figure 4.8 is not quite precise because projections from modules are done here by position—$\pi_1 M$ and $\pi_2 M$ are the first and second

Purity Classifiers	$\kappa ::=$	$\mathtt{P} \mid \mathtt{I}$
Terms	$e ::=$	$\cdots \mid \mathsf{Term}(\mathrm{M}) \mid \mathsf{pack}\ \mathrm{M}\ \mathsf{as}\ \mathrm{S}$
Modules	$\mathrm{M}, \mathrm{N}, \mathrm{F} ::=$	$\mathrm{X} \mid \langle\rangle \mid [\mathrm{C}] \mid [e] \mid \langle \mathrm{X} = \mathrm{M}_1, \mathrm{M}_2 \rangle \mid \pi_i \mathrm{M} \mid$
		$\lambda^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}.\mathrm{M} \mid \mathrm{F}^{\mathsf{tot}}(\mathrm{M}) \mid \lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M} \mid \mathrm{F}^{\mathsf{par}}(\mathrm{M}) \mid$
		$\mathrm{M} :>_\kappa \mathrm{S} \mid \mathsf{purify}(\mathrm{M}) \mid \mathsf{unpack}\ \mathrm{M}\ \mathsf{as}\ \mathrm{S} \mid \mathsf{let}\ \mathrm{X} = \mathrm{M}'\ \mathsf{in}\ (\mathrm{M} : \mathrm{S})$
Projectible Modules	$\mathbb{M}, \mathbb{N}, \mathbb{F} ::=$	$\mathrm{X} \mid \langle\rangle \mid [\mathrm{C}] \mid [e] \mid \langle \mathrm{X} = \mathbb{M}_1, \mathbb{M}_2 \rangle \mid \pi_i \mathbb{M} \mid$
		$\lambda^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}.\mathbb{M} \mid \mathbb{F}^{\mathsf{tot}}(\mathbb{M}) \mid \lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M}$
Dynamic Contexts	$\Gamma ::=$	$\cdots \mid \Gamma, \mathrm{X}{:}\mathrm{S}$

Figure 4.7: Syntax of Modules

$\langle\rangle$	$\approx$	`struct end`
$[\mathrm{C}]$	$\approx$	`struct type t =` C `end`
$[e]$	$\approx$	`struct val x =` e `end`

$\langle \mathrm{X} = \mathrm{M}_1, \mathrm{M}_2 \rangle$	$\approx$	`struct`
		    `structure X =` M_1
		    `structure Y =` M_2
		`end`

$\lambda^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{M}$	$\approx$	`functor (X :` S_1`) ->` M
$\lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M}$	$\approx$	`functor (X :` S_1`) :>` S_2 `->` M

Figure 4.8: Correspondence With ML Modules

projections of M—whereas in ML they are done by name. Correspondingly, X is alpha-convertible in $\langle \mathrm{X} = \mathrm{M}_1, \mathrm{M}_2 \rangle$, but changing X to X′ in the ML analogue results in a module with a different signature. I will sometimes write $\langle \mathrm{M}_1, \mathrm{M}_2 \rangle$ as shorthand when $\mathrm{X} \notin \mathrm{FV}(\mathrm{M}_2)$.

Total functors are written $\lambda^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}.\mathrm{M}$, where X is the argument variable, S the argument signature, and M the functor body. The application of a total functor F to an argument M is written $\mathrm{F}^{\mathsf{tot}}(\mathrm{M})$. The syntax for partial functor introduction and elimination is similar, except that the partial functor introduction form $\lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M}$ also includes a result signature S_2. The result signature is needed in order to ensure that modules have principal signatures. If a result signature were not required, then $\lambda^{\mathsf{par}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{M}$ could be assigned a range of different signatures $\Pi^{\mathsf{par}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$, with S_2 ranging over different signatures for M. Even if M has a principal signature S_M, the signature $\Pi^{\mathsf{par}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_\mathrm{M}$ would not be principal for $\lambda^{\mathsf{par}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{M}$ because partial functor subtyping is invariant in the result signature. In practice, however, this is not an important issue—in the language design I present in Part III, the result signatures of partial functors are simply inferred.

Sealed module expressions are written $\mathrm{M} :>_\kappa \mathrm{S}$, where κ is a purity classifier that can either be `P` for "pure" or `I` for "impure." Recall that we are not differentiating here between (static) purity and separability, so `P` and `I` may also be read as "separable" and "inseparable," respectively. $\mathrm{M} :>_\mathtt{P} \mathrm{S}$ corresponds to the basic form of sealing, while $\mathrm{M} :>_\mathtt{I} \mathrm{S}$ corresponds to the impure form of sealing. The motivation for these two forms was described in Section 2.1.5. As noted in Section 2.1.6, $\mathrm{M} :>_\mathtt{I} \mathrm{S}$ may be encoded via partial functors as follows: $(\lambda^{\mathsf{par}}(\mathrm{X}{:}1){:}\mathrm{S}.\mathrm{M})(\langle\rangle)$. I am not aware of any similar, purely syntactic encoding of basic sealing, although Shan [67] has described a rather complex global program transformation that effectively translates uses of basic sealing into uses of impure sealing.

As I explained in Section 2.1.7, modules that can be given transparent signatures should be considered pure. For simplicity, however, my module typing judgment implements this semantics somewhat lazily: it will not necessarily consider all transparent modules to be pure, but if one wants a transparent module M to be considered pure, one can indicate this by writing $\mathsf{purify}(\mathrm{M})$. The purify expression has no run-time effect, it merely forces the module to be considered pure.

The new term-level construct $\mathsf{pack}\ \mathrm{M}\ \mathsf{as}\ \mathrm{S}$ and module-level construct $\mathsf{unpack}\ e\ \mathsf{as}\ \mathrm{S}$ serve as coercions between modules and terms. The former packages a module M of signature S as a term of package type $\langle\!|\mathrm{S}|\!\rangle$, and the latter unpackages a term e of type $\langle\!|\mathrm{S}|\!\rangle$ into a module of signature S.[7] This enables support for first-class module programming when desired.

Lastly, I also include a let-expression at the module level, $\mathsf{let}\ \mathrm{X} = \mathrm{M}_1\ \mathsf{in}\ (\mathrm{M}_2 : \mathrm{S})$. The signature annotation S is present to ensure that the let module has a principal signature. (This issue will be discussed further in Section 4.2.6.) The type system treats $\mathsf{let}\ \mathrm{X} = \mathrm{M}_1\ \mathsf{in}\ (\mathrm{M}_2 : \mathrm{S})$ as if its body contained an instance of basic sealing (*i.e.,* $\mathrm{M}_2 \mathrel{\texttt{:>}_{\mathsf{P}}} \mathrm{S}$). I will sometimes omit the S when it is obvious from context what it should be.

4.2.2 Projectible Modules

Figure 4.7 also defines a syntactic subclass of *projectible* modules, written $\mathbb{M}$. The criterion here for projectibility is based on the analysis of Chapter 2: projectible modules are essentially those modules which are pure/separable and which do not contain any uses of sealing. While this description is mostly accurate, it does not tell the complete story.

First of all, I only treat as projectible those modules which the type system considers pure *without* the aid of purify coercions. The reason for this is primarily technical: I want to be able to extract the static part of a projectible module—a constructor comprising the module's type components—in a purely syntactic, context-free manner, and for certain *morally* projectible modules this is not possible. For example, suppose that X is a variable bound in the context with a partial functor signature $\Pi^{\mathsf{par}}\mathrm{Y}{:}\mathrm{S}_1.\mathrm{S}_2$, where S_2 is transparent. Applying X to some projectible module $\mathbb{M}$ results in a transparent module, which is morally projectible. However, there is no way to tell that $\mathrm{X}^{\mathsf{par}}(\mathbb{M})$ is projectible (or even pure), and certainly no way to extract its static part, without knowing the signature that X is bound with. I argue that not being able to project types from transparent modules like $\mathrm{X}^{\mathsf{par}}(\mathbb{M})$ does not result in any fundamental loss of expressiveness anyway—for any type component `t` in a transparent module N, the signature of N will contain a specification `type t =` C that indicates a well-formed type C to which N`.t` is equivalent.

Second, what does it mean for a *functor* to be projectible? The answer is that the definition of projectibility in this calculus is a bit more general than the definition given at the start of Chapter 2: a module is considered projectible in this language if one may *extract* types from it, not necessarily by direct projection. In the case of a functor, extracting types means first applying the functor and then projecting (or extracting) types from the result.[8] Hence, total functors are projectible so long as their bodies are projectible, and total functor applications are projectible so long as the functor and the argument are both projectible.

Given this generalization of projectibility, one may be surprised to find that *all* partial functor

[7]The package type $\langle\!|\mathrm{S}|\!\rangle$ was defined above, in Figure 4.6, in terms of the existing type structure of the core language.

[8]Perhaps "extractability" would have been a better word to use than "projectibility" from the beginning, but the latter has the advantage of being the same term that we used in Dreyer *et al.* [12].

$$
\begin{array}{lcl}
\mathsf{Fst}(\mathrm{X}) & \stackrel{\mathrm{def}}{=} & \mathrm{X}^{\mathsf{c}} \\
\mathsf{Fst}(\langle\rangle) & \stackrel{\mathrm{def}}{=} & \langle\rangle \\
\mathsf{Fst}([\mathrm{C}]) & \stackrel{\mathrm{def}}{=} & \mathrm{C} \\
\mathsf{Fst}([e]) & \stackrel{\mathrm{def}}{=} & \langle\rangle \\
\mathsf{Fst}(\langle \mathrm{X} = \mathbb{M}_1, \mathbb{M}_2\rangle) & \stackrel{\mathrm{def}}{=} & \langle \mathrm{X}^{\mathsf{c}} = \mathsf{Fst}(\mathbb{M}_1), \mathsf{Fst}(\mathbb{M}_2)\rangle \\
\mathsf{Fst}(\pi_i \mathbb{M}) & \stackrel{\mathrm{def}}{=} & \pi_i(\mathsf{Fst}(\mathbb{M})) \\
\mathsf{Fst}(\lambda^{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}.\mathbb{M}) & \stackrel{\mathrm{def}}{=} & \lambda \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}).\mathsf{Fst}(\mathbb{M}) \\
\mathsf{Fst}(\mathbb{F}^{\mathsf{tot}}(\mathbb{M})) & \stackrel{\mathrm{def}}{=} & \mathsf{Fst}(\mathbb{F})(\mathsf{Fst}(\mathbb{M})) \\
\mathsf{Fst}(\lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M}) & \stackrel{\mathrm{def}}{=} & \langle\rangle
\end{array}
$$

Figure 4.9: Extracting the Static Part of a Projectible Module

expressions $\lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M}$ are considered projectible. The reason for this is simple. The type system will never allow a partial functor expression to be applied inside a pure module, let alone a projectible module, so any appearance that a partial functor makes inside a pure/projectible module expression will be useless as far as the extraction of type components is concerned. Since partial functors can only make useless appearances inside projectible modules, there is no good reason to banish them.[9] The same goes for the unit module $\langle\rangle$ and term module $[e]$, neither of which has any type components that could be extracted. Note that these three module forms—unit, term and partial functor—are precisely those which inhabit the "unitary" signatures S for which $\mathsf{Fst}(\mathrm{S})$ is the unit kind.

As explained in Section 2.2.3, I have chosen in this calculus not to introduce any new type-level construct for projecting types from modules. Instead, I define a meta-level function $\mathsf{Fst}(\mathbb{M})$ (shown in Figure 4.9) that computes a type constructor representing the static part of $\mathbb{M}$. The type components of $\mathbb{M}$ may then be projected (or, more generally, extracted) from $\mathsf{Fst}(\mathbb{M})$, rather than from $\mathbb{M}$ itself. Given the above discussion, the definition of $\mathsf{Fst}(\mathbb{M})$ should be fairly self-explanatory: the only unusual cases are the modules $\mathbb{M}$ for which $\mathsf{Fst}(\mathbb{M})$ is always unit, and these are precisely the modules from which no type components can be extracted.

The definition of $\mathsf{Fst}(\mathbb{M})$ ensures that, when checking equivalence of types projected from modules, arguments to total functors are compared via *static* equivalence. For example, supposing that the result signature of a functor $\mathbb{F}$ were $[\![\mathbf{T}]\!]$, the types $\mathsf{Fst}(\mathbb{F}^{\mathsf{tot}}(\mathbb{M}_1))$ and $\mathsf{Fst}(\mathbb{F}^{\mathsf{tot}}(\mathbb{M}_2))$ will be equivalent whenever $\mathsf{Fst}(\mathbb{M}_1)$ and $\mathsf{Fst}(\mathbb{M}_2)$ are equivalent, *i.e.*, whenever $\mathbb{M}_1$ and $\mathbb{M}_2$ have equivalent type components.

[9] For those familiar with the previous version of this type system (Dreyer, Crary and Harper [12], described in Section 2.3): it is worth noting here that the desire to treat partial functor expressions as projectible is the only motivation for why DCH's strange "S" purity class exists. In the DCH type system, a module is only considered projectible if it is pure *and* free of any sealing. Thus, while DCH considers all functors to be pure (because they are values), it considers a partial functor $\lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M}$ to be projectible only if the body M is also free of sealing. However, since this is a partial functor we are talking about, M is still permitted to be impure. The "S" purity class describes precisely those modules that are potentially impure but free of sealing, so $\lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M}$ is projectible in DCH if and only if M can be given purity classification S. My present approach, which is to just treat all partial functor expressions as projectible, seems much simpler.

4.2.3 Static Semantics

Figure 4.10 shows the inference rules for the judgment of well-formed modules, as well as new rules for the judgments of well-formed terms and dynamic contexts. The module judgment $\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}$ says that M has signature S and purity κ. If $\kappa = \mathtt{P}$, then M is pure; if $\kappa = \mathtt{I}$, then M may be impure. I will sometimes make use of meets ($\sqcap$) and joins ($\sqcup$) in the two-point lattice $\mathtt{P} \sqsubseteq \mathtt{I}$.

As I have extended the syntax of dynamic contexts, I will also extend the erasure function mapping dynamic contexts to static contexts as follows:

$$\mathsf{Fst}(\Gamma, \mathrm{X}{:}\mathrm{S}) \quad \stackrel{\text{def}}{=} \quad \mathsf{Fst}(\Gamma), \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S})$$

As in the term well-formedness judgment of Chapter 3, some of the inference rules in Figure 4.10 refer in their premises to the judgments involving constructors, kinds and signatures, except with a dynamic context Γ in place of a static context Δ. The meaning of these premises is the same one given before by Definition 3.2.1, namely that $\Gamma \vdash \mathcal{J}$ is shorthand for the conjunction of $\Gamma \vdash \mathrm{ok}$ and $\mathsf{Fst}(\Gamma) \vdash \mathcal{J}$.

Some notational conveniences: Since module variables X cannot appear directly in signatures (only their static parts X^{c} can), $\mathrm{S}[\mathbb{M}/\mathrm{X}]$ should be taken as shorthand for $\mathrm{S}[\mathsf{Fst}(\mathbb{M})/\mathrm{X}^{\mathsf{c}}]$. Similarly, the notation $\mathfrak{S}_{\mathrm{S}}(\mathbb{M})$ is shorthand for the signature $\mathfrak{S}_{\mathrm{S}}(\mathsf{Fst}(\mathbb{M}))$, which classifies precisely those modules of signature S which are statically equivalent to $\mathbb{M}$, *i.e.,* whose static parts are equivalent to $\mathsf{Fst}(\mathbb{M})$.

Now for the rules, many of which should appear very similar to the rules for well-formed constructors. Rules 84–90 are completely straightforward. For Rule 90, recall that the purity of $[e]$ relates to the module's lack of type components and does not imply anything about the computational purity of the term e. Rules 91, 92 and 101 say that pairs, first projections and let-expressions are as pure as their component modules.

Rule 93, however, requires that second projections only be made from *projectible* (and thus pure) modules. The reason for this restriction is that the signature of $\pi_2\mathrm{M}$ is formed by substituting $\mathsf{Fst}(\pi_1\mathrm{M})$ in for X^{c} in the signature S'' of M's second component. $\mathsf{Fst}(\pi_1\mathrm{M})$ is only a valid operation when $\pi_1\mathrm{M}$ is a projectible module, which in turn implies that M itself must be projectible. One potential way to ameliorate this restriction would be to require that the signature of M be a non-dependent pair signature $\mathrm{S}' \times \mathrm{S}''$. The signature of $\pi_2\mathrm{M}$ would then be simply S'', requiring no substitutions. Unfortunately, as I will explain in Section 4.2.6, this approach runs afoul of something called the "avoidance problem." In practice, though, this restriction does not seem to be a major issue since all existing ML dialects impose it.

Rules 94 and 96 say that total and partial functors are pure expressions (of course, because they are values), but that total functors must in addition have pure bodies. Rule 95 says that a total functor application $\mathrm{F}^{\mathsf{tot}}(\mathrm{M})$ is as pure as F and M are, whereas Rule 97 says that partial functor applications $\mathrm{F}^{\mathsf{par}}(\mathrm{M})$ are always considered impure. Note that in both cases the functor argument M is required to be projectible, for the same reason that second projections are only permitted from projectible modules. The signature of $\mathrm{F}^{\tau}(\mathrm{M})$ requires $\mathsf{Fst}(\mathrm{M})$ to be substituted for X^{c} in the result signature S'', which is only valid if M is projectible.

Rule 98 joins the rules for basic and impure sealing into one. When $\kappa = \mathtt{P}$, the sealed module $\mathrm{M} \mathbin{\texttt{:>}_\kappa} \mathrm{S}$ has the same purity as M; when $\kappa = \mathtt{I}$, the sealed module will be impure regardless. In either case, the sealed module does not belong to the syntax of projectible module expressions. Rule 99 says that $\mathsf{purify}(\mathrm{M})$ is well-formed and pure, regardless of M's inferred purity level, so long as M can be given a transparent signature.

Rule 100 says that unpacking a term of package type into a module expression results in a potentially impure module, because the type components of the module may depend on core-

New rules for well-formed dynamic contexts: $\Gamma \vdash \mathrm{ok}$

$$\frac{\Gamma \vdash \mathrm{S}\ \mathrm{sig}}{\Gamma, \mathrm{X}{:}\mathrm{S} \vdash \mathrm{ok}}\ (84)$$

New rules for well-formed terms: $\Gamma \vdash e : \mathrm{C}$

$$\frac{\Gamma \vdash \mathrm{M} :_\kappa [\![\mathrm{C}]\!]}{\Gamma \vdash \mathsf{Term}(\mathrm{M}) : \mathrm{C}}\ (85) \qquad \frac{\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}}{\Gamma \vdash \mathsf{pack}\ \mathrm{M}\ \mathsf{as}\ \mathrm{S} : \langle\!|\mathrm{S}|\!\rangle}\ (86)$$

Well-formed modules: $\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}$

$$\frac{\Gamma \vdash \mathrm{ok} \quad \mathrm{X}{:}\mathrm{S} \in \Gamma}{\Gamma \vdash \mathrm{X} :_{\mathtt{P}} \mathrm{S}}\ (87) \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \langle\rangle :_{\mathtt{P}} 1}\ (88) \qquad \frac{\Gamma \vdash \mathrm{C} : \mathrm{K}}{\Gamma \vdash [\mathrm{C}] :_{\mathtt{P}} [\![\mathrm{K}]\!]}\ (89) \qquad \frac{\Gamma \vdash e : \mathrm{C}}{\Gamma \vdash [e] :_{\mathtt{P}} [\![\mathrm{C}]\!]}\ (90)$$

$$\frac{\Gamma \vdash \mathrm{M}' :_\kappa \mathrm{S}' \quad \Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M}'' :_\kappa \mathrm{S}''}{\Gamma \vdash \langle \mathrm{X} = \mathrm{M}', \mathrm{M}'' \rangle :_\kappa \Sigma \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''}\ (91) \qquad \frac{\Gamma \vdash \mathrm{M} :_\kappa \Sigma \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''}{\Gamma \vdash \pi_1 \mathrm{M} :_\kappa \mathrm{S}'}\ (92) \qquad \frac{\Gamma \vdash \mathbb{M} :_{\mathtt{P}} \Sigma \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''}{\Gamma \vdash \pi_2 \mathbb{M} :_{\mathtt{P}} \mathrm{S}''[\pi_1 \mathbb{M}/\mathrm{X}]}\ (93)$$

$$\frac{\Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M} :_{\mathtt{P}} \mathrm{S}''}{\Gamma \vdash \lambda^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}'.\mathrm{M} :_{\mathtt{P}} \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''}\ (94) \qquad \frac{\Gamma \vdash \mathrm{F} :_\kappa \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}'' \quad \Gamma \vdash \mathbb{M} :_{\mathtt{P}} \mathrm{S}'}{\Gamma \vdash \mathrm{F}^{\mathsf{tot}}(\mathbb{M}) :_\kappa \mathrm{S}''[\mathbb{M}/\mathrm{X}]}\ (95)$$

$$\frac{\Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M} :_{\mathtt{I}} \mathrm{S}''}{\Gamma \vdash \lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}'){:}\mathrm{S}''.\mathrm{M} :_{\mathtt{P}} \Pi^{\mathsf{par}} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''}\ (96) \qquad \frac{\Gamma \vdash \mathrm{F} :_\kappa \Pi^{\mathsf{par}} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}'' \quad \Gamma \vdash \mathbb{M} :_{\mathtt{P}} \mathrm{S}'}{\Gamma \vdash \mathrm{F}^{\mathsf{par}}(\mathbb{M}) :_{\mathtt{I}} \mathrm{S}''[\mathbb{M}/\mathrm{X}]}\ (97)$$

$$\frac{\Gamma \vdash \mathrm{M} :_{\kappa'} \mathrm{S}}{\Gamma \vdash (\mathrm{M} :>_\kappa \mathrm{S}) :_{\kappa \sqcup \kappa'} \mathrm{S}}\ (98) \qquad \frac{\Gamma \vdash \mathrm{M} :_\kappa \mathbb{S}}{\Gamma \vdash \mathsf{purify}(\mathrm{M}) :_{\mathtt{P}} \mathbb{S}}\ (99)$$

$$\frac{\Gamma \vdash e : \langle\!|\mathrm{S}|\!\rangle \quad \Gamma \vdash \mathrm{S}\ \mathrm{sig}}{\Gamma \vdash \mathsf{unpack}\ e\ \mathsf{as}\ \mathrm{S} :_{\mathtt{I}} \mathrm{S}}\ (100) \qquad \frac{\Gamma \vdash \mathrm{M}' :_\kappa \mathrm{S}' \quad \Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M} :_\kappa \mathrm{S} \quad \Gamma \vdash \mathrm{S}\ \mathrm{sig}}{\Gamma \vdash \mathsf{let}\ \mathrm{X} = \mathrm{M}'\ \mathsf{in}\ (\mathrm{M} : \mathrm{S}) :_\kappa \mathrm{S}}\ (101)$$

$$\frac{\Gamma \vdash \mathbb{M} :_{\mathtt{P}} \mathrm{S}}{\Gamma \vdash \mathbb{M} :_{\mathtt{P}} \mathfrak{S}_{\mathrm{S}}(\mathbb{M})}\ (102) \qquad \frac{\Gamma \vdash \mathrm{M} :_{\mathtt{P}} \mathrm{S}}{\Gamma \vdash \mathrm{M} :_{\mathtt{I}} \mathrm{S}}\ (103) \qquad \frac{\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}' \quad \Gamma \vdash \mathrm{S}' \leq \mathrm{S}}{\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}}\ (104)$$

Figure 4.10: Inference Rules for Modules

language computational effects. Note that the second premise of the rule is needed because the well-formedness of the type $\langle\!|\mathrm{S}|\!\rangle$ does not necessarily imply that S is a well-formed signature.

Rule 102 implements selfification, also known as signature strengthening. It says that a projectible module $\mathbb{M}$ with signature S may also be assigned the (potentially) more precise signature $\mathfrak{S}_{\mathrm{S}}(\mathbb{M})$. This rule is critical to ensuring principal signatures for modules. It is primarily useful when M is a variable X. For example, if X has signature $\Sigma\mathrm{Y}{:}[\![\mathbf{T}]\!].[\![\mathrm{Y}^{\mathsf{c}}]\!]$, which corresponds to the ML signature `sig type t ; val v : t end`, then the selfification rule assigns X the more precise $[\![\mathfrak{S}(\pi_1 \mathrm{X}^{\mathsf{c}})]\!] \times [\![\pi_1 \mathrm{X}^{\mathsf{c}}]\!]$, which corresponds in ML to `sig type t = X.t ; val v : X.t end`. This "selfified" signature encapsulates all that is known statically about X.

Finally, Rule 103 allows the purity of a module to be forgotten, and Rule 104 implements subsumption.

4.2.4 Declarative Properties

It is easy to check that the declarative properties of the core language continue to hold under the extension of dynamic contexts with module variable bindings, and that all the structural properties except Substitution apply to the module well-formedness judgment as well. Additionally, we have the following new properties:

Proposition 4.2.1 (Subderivations)
Every proof of $\Gamma_1, \mathrm{X}{:}\mathrm{S}, \Gamma_2 \vdash \mathcal{J}$ contains a strict subderivation of $\Gamma_1 \vdash \mathrm{S}\ \mathrm{sig}$.

Proposition 4.2.2 (Weakening)
If $\Gamma_1, \mathrm{X}{:}\mathrm{S}_2, \Gamma_2 \vdash \mathcal{J}$ and $\Gamma_1 \vdash \mathrm{S}_1 \leq \mathrm{S}_2$, then $\Gamma_1, \mathrm{X}{:}\mathrm{S}_1, \Gamma_2 \vdash \mathcal{J}$.

Proposition 4.2.3 ($\mathsf{Fst}(\mathbb{M})$ Preserves Well-Formedness)
If $\Gamma \vdash \mathbb{M} :_\kappa \mathrm{S}$, then $\Gamma \vdash \mathbb{M} :_{\mathrm{P}} \mathrm{S}$ and $\Gamma \vdash \mathsf{Fst}(\mathbb{M}) : \mathsf{Fst}(\mathrm{S})$.

Proof: By induction on derivations. ■

Proposition 4.2.4 (Validity)
If $\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}$, then $\Gamma \vdash \mathrm{S}\ \mathrm{sig}$.

Proof: By induction on derivations, with straightforward uses of Proposition 4.2.3. ■

4.2.5 Signature Checking and Synthesis

Figure 4.11 defines a signature checking algorithm for modules. To check whether a module M has a given signature S and purity level κ, the algorithm synthesizes the principal signature S' and minimal purity level κ' of M, and checks whether S' and κ' are smaller than S and κ, respectively. Like the kind synthesis algorithm, the signature synthesis algorithm is very similar to the declarative system, except that (1) it expects the context it is given to be well-formed, (2) it only selfifies variables, and (3) it restricts the use of subsumption to functor arguments and sealed modules.

Here I give several properties of signature synthesis, including that it is sound and complete, and that the principal signatures of projectible modules are always transparent. This last fact comes in handy in proving Part 3 of the completeness theorem.

Proposition 4.2.5 (Soundness and Other Properties of Signature Checking/Synthesis)
Assume $\Gamma \vdash \mathrm{ok}$.

1. If $\Gamma \vdash \mathrm{M} \Rightarrow_\kappa \mathrm{S}$ or $\Gamma \vdash \mathrm{M} \Leftarrow_\kappa \mathrm{S}$, then $\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}$.

2. If $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa_1} \mathrm{S}_1$ and $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa_2} \mathrm{S}_2$, then $\mathrm{S}_1 = \mathrm{S}_2$ and $\kappa_1 = \kappa_2$.

3. If $\Gamma \vdash \mathbb{M} \Rightarrow_{\mathrm{P}} \mathrm{S}$, then S is transparent.

4. For $\mathcal{J}$ ranging over any judgment defined in Figure 4.11,
   if $\Gamma \vdash \mathcal{J}$ and $\Gamma' \supseteq \Gamma$ and $\Gamma' \vdash \mathrm{ok}$, then $\Gamma' \vdash \mathcal{J}$.

Proof: By straightforward induction on the algorithm. ■

Signature checking: $\Gamma \vdash \mathrm{M} \Leftarrow_{\kappa} \mathrm{S}$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa'} \mathrm{S}' \quad \Gamma \vdash \mathrm{S}' \leq \mathrm{S} \quad \kappa' \sqsubseteq \kappa}{\Gamma \vdash \mathrm{M} \Leftarrow_{\kappa} \mathrm{S}}$$

Signature synthesis: $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \mathrm{S}$

$$\frac{\mathrm{X}{:}\mathrm{S} \in \Gamma}{\Gamma \vdash \mathrm{X} \Rightarrow_{\mathtt{P}} \mathfrak{S}_{\mathrm{S}}(\mathrm{X})} \qquad \frac{}{\Gamma \vdash \langle\rangle \Rightarrow_{\mathtt{P}} 1} \qquad \frac{\Gamma \vdash \mathrm{C} \Rightarrow \mathrm{K}}{\Gamma \vdash [\mathrm{C}] \Rightarrow_{\mathtt{P}} [\![\mathrm{K}]\!]} \qquad \frac{\Gamma \vdash e \Rightarrow \mathrm{C}}{\Gamma \vdash [e] \Rightarrow_{\mathtt{P}} [\![\mathrm{C}]\!]}$$

$$\frac{\Gamma \vdash \mathrm{M}' \Rightarrow_{\kappa'} \mathrm{S}' \quad \Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M}'' \Rightarrow_{\kappa''} \mathrm{S}''}{\Gamma \vdash \langle \mathrm{X} = \mathrm{M}', \mathrm{M}'' \rangle \Rightarrow_{\kappa' \sqcup \kappa''} \Sigma \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''} \qquad \frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \Sigma \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''}{\Gamma \vdash \pi_1 \mathrm{M} \Rightarrow_{\kappa} \mathrm{S}'} \qquad \frac{\Gamma \vdash \mathbb{M} \Rightarrow_{\mathtt{P}} \Sigma \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''}{\Gamma \vdash \pi_2 \mathbb{M} \Rightarrow_{\mathtt{P}} \mathrm{S}''[\pi_1 \mathbb{M}/\mathrm{X}]}$$

$$\frac{\Gamma \vdash \mathrm{S}' \text{ sig} \quad \Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M} \Rightarrow_{\mathtt{P}} \mathrm{S}''}{\Gamma \vdash \lambda^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}'.\mathrm{M} \Rightarrow_{\mathtt{P}} \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''} \qquad \frac{\Gamma \vdash \mathrm{F} \Rightarrow_{\kappa} \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}'' \quad \Gamma \vdash \mathbb{M} \Leftarrow_{\mathtt{P}} \mathrm{S}'}{\Gamma \vdash \mathrm{F}^{\mathsf{tot}}(\mathbb{M}) \Rightarrow_{\kappa} \mathrm{S}''[\mathbb{M}/\mathrm{X}]}$$

$$\frac{\Gamma \vdash \mathrm{S}' \text{ sig} \quad \Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M} \Leftarrow_{\mathtt{I}} \mathrm{S}''}{\Gamma \vdash \lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}'){:}\mathrm{S}''.\mathrm{M} \Rightarrow_{\mathtt{P}} \Pi^{\mathsf{par}} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}''} \qquad \frac{\Gamma \vdash \mathrm{F} \Rightarrow_{\kappa} \Pi^{\mathsf{par}} \mathrm{X}{:}\mathrm{S}'.\mathrm{S}'' \quad \Gamma \vdash \mathbb{M} \Leftarrow_{\mathtt{P}} \mathrm{S}'}{\Gamma \vdash \mathrm{F}^{\mathsf{par}}(\mathbb{M}) \Rightarrow_{\mathtt{I}} \mathrm{S}''[\mathbb{M}/\mathrm{X}]}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa'} \mathrm{S}' \quad \Gamma \vdash \mathrm{S}' \leq \mathrm{S}}{\Gamma \vdash (\mathrm{M} :>_{\kappa} \mathrm{S}) \Rightarrow_{\kappa \sqcup \kappa'} \mathrm{S}} \qquad \frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \mathbb{S}}{\Gamma \vdash \mathsf{purify}(\mathrm{M}) \Rightarrow_{\mathtt{P}} \mathbb{S}} \qquad \frac{\Gamma \vdash \mathrm{S} \text{ sig} \quad \Gamma \vdash e \Leftarrow \langle\!|\mathrm{S}|\!\rangle}{\Gamma \vdash \mathsf{unpack}\ e\ \mathsf{as}\ \mathrm{S} \Rightarrow_{\mathtt{I}} \mathrm{S}}$$

$$\frac{\Gamma \vdash \mathrm{M}' \Rightarrow_{\kappa'} \mathrm{S}' \quad \Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M} \Rightarrow_{\kappa''} \mathrm{S}'' \quad \Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{S}'' \leq \mathrm{S} \quad \Gamma \vdash \mathrm{S} \text{ sig}}{\Gamma \vdash \mathsf{let}\ \mathrm{X} = \mathrm{M}'\ \mathsf{in}\ (\mathrm{M} : \mathrm{S}) \Rightarrow_{\kappa' \sqcup \kappa''} \mathrm{S}}$$

New type synthesis rules: $\Gamma \vdash e \Rightarrow \mathrm{C}$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} [\![\mathrm{C}]\!]}{\Gamma \vdash \mathsf{Term}(\mathrm{M}) \Rightarrow \mathrm{C}} \qquad \frac{\Gamma \vdash \mathrm{M} \Leftarrow_{\kappa} \mathrm{S}}{\Gamma \vdash \mathsf{pack}\ \mathrm{M}\ \mathsf{as}\ \mathrm{S} \Rightarrow \langle\!|\mathrm{S}|\!\rangle}$$

Figure 4.11: Signature Checking and Principal Signature Synthesis

Theorem 4.2.6 (Completeness of Signature Checking)

1. If $\Gamma \vdash e : \mathrm{C}$, then $\Gamma \vdash e \Leftarrow \mathrm{C}$.
2. If $\Gamma \vdash \mathrm{M} :_{\kappa} \mathrm{S}$, then $\Gamma \vdash \mathrm{M} \Leftarrow_{\kappa} \mathrm{S}$.
3. If $\Gamma \vdash \mathbb{M} :_{\mathtt{P}} \mathrm{S}$, then $\Gamma \vdash \mathbb{M} \Leftarrow_{\mathtt{P}} \mathfrak{S}_{\mathrm{S}}(\mathbb{M})$.

Proof: By induction, first on the structure of the module M (or term e), and second on the structure of the derivation of the premise. The proof of Parts 1 and 2 are by cases on the structure of the input derivation and are completely straightforward, but I will write out most of the cases anyway (see below). There are only two cases, Rules 91 and 101, in which induction is applied to a derivation that is not a subderivation of the input derivation. It is these cases, however, that necessitate induction first on the structure of M.

The proof of Part 3 follows easily from Part 2: If $\Gamma \vdash \mathbb{M} :_{\mathtt{P}} \mathrm{S}$, then by Part 2, $\Gamma \vdash \mathbb{M} \Rightarrow_{\mathtt{P}} \mathbb{R}$, where $\Gamma \vdash \mathbb{R} \leq \mathrm{S}$ (and $\mathbb{R}$ is transparent due to Part 3 of Proposition 4.2.5). By Soundness, $\Gamma \vdash \mathbb{M} :_{\mathtt{P}} \mathbb{R}$. Then, by Proposition 4.1.9, $\Gamma \vdash \mathbb{R} \equiv \mathfrak{S}_{\mathbb{R}}(\mathbb{M})$ and $\Gamma \vdash \mathfrak{S}_{\mathbb{R}}(\mathbb{M}) \leq \mathfrak{S}_{\mathrm{S}}(\mathbb{M})$. Thus, by Transitivity, $\Gamma \vdash \mathbb{R} \leq \mathfrak{S}_{\mathrm{S}}(\mathbb{M})$ as desired.

- Case: Rules 55–67. Same as in proof of Proposition 3.2.6.

- Case: Rules 85–88. Straightforward.
- Case: Rule 89. By Proposition 3.1.16.
- Case: Rule 90. Straightforward.
- Case: Rule 91.
  1. By induction, $\Gamma \vdash \mathrm{M}' \Rightarrow_{\kappa'} \mathrm{R}'$, where $\Gamma \vdash \mathrm{R}' \leq \mathrm{S}'$ and $\kappa' \sqsubseteq \kappa$.
  2. By Weakening, $\Gamma, \mathrm{X{:}R}' \vdash \mathrm{M}'' :_\kappa \mathrm{S}''$.
  3. By induction, $\Gamma, \mathrm{X{:}R}' \vdash \mathrm{M}'' \Rightarrow_{\kappa''} \mathrm{R}''$, where $\Gamma, \mathrm{X{:}R}' \vdash \mathrm{R}'' \leq \mathrm{S}''$ and $\kappa'' \sqsubseteq \kappa$.
  4. Thus, $\Gamma \vdash \langle \mathrm{X} = \mathrm{M}', \mathrm{M}'' \rangle \Rightarrow_{\kappa' \sqcup \kappa''} \Sigma \mathrm{X{:}R}'.\mathrm{R}''$,
  5. and $\Gamma \vdash \Sigma \mathrm{X{:}R}'.\mathrm{R}'' \leq \Sigma \mathrm{X{:}S}'.\mathrm{S}''$, and $\kappa' \sqcup \kappa'' \sqsubseteq \kappa$.
- Case: Rule 92.
  1. By induction, $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa'} \Sigma \mathrm{X{:}R}'.\mathrm{R}''$, where $\Gamma \vdash \mathrm{R}' \leq \mathrm{S}'$, $\Gamma, \mathrm{X{:}R}' \vdash \mathrm{R}'' \leq \mathrm{S}''$ and $\kappa' \sqsubseteq \kappa$.
  2. Thus, $\Gamma \vdash \pi_1 \mathrm{M} \Rightarrow_{\kappa'} \mathrm{R}'$.
- Case: Rule 93.
  1. By induction, $\Gamma \vdash \mathbb{M} \Rightarrow_{\mathsf{P}} \Sigma \mathrm{X{:}R}'.\mathrm{R}''$, where $\Gamma \vdash \mathrm{R}' \leq \mathrm{S}'$ and $\Gamma, \mathrm{X{:}R}' \vdash \mathrm{R}'' \leq \mathrm{S}''$.
  2. Thus, $\Gamma \vdash \pi_1 \mathbb{M} :_{\mathsf{P}} \mathrm{R}'$ and $\Gamma \vdash \pi_2 \mathbb{M} \Rightarrow_{\mathsf{P}} \mathrm{R}''[\pi_1 \mathbb{M}/\mathrm{X}]$.
  3. By Substitution, $\Gamma \vdash \mathrm{R}''[\pi_1 \mathbb{M}/\mathrm{X}] \leq \mathrm{S}''[\pi_1 \mathbb{M}/\mathrm{X}]$.
- Case: Rule 94.
  1. By induction, $\Gamma, \mathrm{X{:}S}' \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{R}''$, where $\Gamma, \mathrm{X{:}S}' \vdash \mathrm{R}'' \leq \mathrm{S}''$.
  2. Thus, $\Gamma \vdash \lambda^{\mathsf{tot}} \mathrm{X{:}S}'.\mathrm{M} \Rightarrow_{\mathsf{P}} \Pi^{\mathsf{tot}} \mathrm{X{:}S}'.\mathrm{R}''$, and $\Gamma \vdash \Pi^{\mathsf{tot}} \mathrm{X{:}S}'.\mathrm{R}'' \leq \Pi^{\mathsf{tot}} \mathrm{X{:}S}'.\mathrm{S}''$.
- Case: Rule 95.
  1. By induction, $\Gamma \vdash \mathrm{F} \Rightarrow_{\kappa'} \Pi^{\mathsf{tot}} \mathrm{X{:}R}'.\mathrm{R}''$, where $\Gamma \vdash \mathrm{R}' \equiv \mathrm{S}'$, $\Gamma, \mathrm{X{:}R}' \vdash \mathrm{R}'' \leq \mathrm{S}''$ and $\kappa' \sqsubseteq \kappa$.
  2. By induction, $\Gamma \vdash \mathbb{M} \Leftarrow_{\mathsf{P}} \mathrm{S}'$ and so $\Gamma \vdash \mathbb{M} \Leftarrow_{\mathsf{P}} \mathrm{R}'$.
  3. Thus, $\Gamma \vdash \mathrm{F}^{\mathsf{tot}}(\mathbb{M}) \Rightarrow_{\kappa'} \mathrm{R}''[\mathbb{M}/\mathrm{X}]$, and by Substitution, $\Gamma \vdash \mathrm{R}''[\mathbb{M}/\mathrm{X}] \leq \mathrm{S}''[\mathbb{M}/\mathrm{X}]$.
- Case: Rule 96.
  1. By induction, $\Gamma, \mathrm{X{:}S}' \vdash \mathrm{M} \Leftarrow_{\mathsf{I}} \mathrm{S}''$.
  2. Thus, $\Gamma \vdash \lambda^{\mathsf{par}}(\mathrm{X{:}S}'){:}\mathrm{S}''.\mathrm{M} \Rightarrow_{\mathsf{P}} \Pi^{\mathsf{par}} \mathrm{X{:}S}'.\mathrm{S}''$.
- Case: Rule 97.
  1. By induction, $\Gamma \vdash \mathrm{F} \Rightarrow_{\kappa'} \Pi^{\mathsf{par}} \mathrm{X{:}R}'.\mathrm{R}''$, where $\Gamma \vdash \mathrm{R}' \equiv \mathrm{S}'$, $\Gamma, \mathrm{X{:}R}' \vdash \mathrm{R}'' \equiv \mathrm{S}''$ and $\kappa' \sqsubseteq \kappa$.
  2. By induction, $\Gamma \vdash \mathbb{M} \Leftarrow_{\mathsf{P}} \mathrm{S}'$ and so $\Gamma \vdash \mathbb{M} \Leftarrow_{\mathsf{P}} \mathrm{R}'$.
  3. Thus, $\Gamma \vdash \mathrm{F}^{\mathsf{par}}(\mathbb{M}) \Rightarrow_{\mathsf{I}} \mathrm{R}''[\mathbb{M}/\mathrm{X}]$, and by Substitution, $\Gamma \vdash \mathrm{R}''[\mathbb{M}/\mathrm{X}] \equiv \mathrm{S}''[\mathbb{M}/\mathrm{X}]$.
- Case: Rule 98.
  1. By induction, $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa''} \mathrm{R}$, where $\Gamma \vdash \mathrm{R} \leq \mathrm{S}$ and $\kappa'' \sqsubseteq \kappa'$.

2. Thus, $\Gamma \vdash \mathrm{M} :>_{\kappa} \mathrm{S} \Rightarrow_{\kappa \sqcup \kappa''} \mathrm{S}$, and $\kappa \sqcup \kappa'' \sqsubseteq \kappa \sqcup \kappa'$.

- Case: Rule 98.

1. By induction, $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa'} \mathrm{R}$, where $\Gamma \vdash \mathrm{R} \leq \mathbb{S}$.
2. By Proposition 4.1.9, R is transparent.
3. Thus, $\Gamma \vdash \mathsf{purify}(\mathrm{M}) \Rightarrow_{\mathsf{P}} \mathrm{R}$.

- Case: Rule 100. Straightforward.

- Case: Rule 101.

1. By induction, $\Gamma \vdash \mathrm{M}' \Rightarrow_{\kappa'} \mathrm{R}'$, where $\Gamma \vdash \mathrm{R}' \leq \mathrm{S}'$ and $\kappa' \sqsubseteq \kappa$.
2. By Weakening, $\Gamma, \mathrm{X}{:}\mathrm{R}' \vdash \mathrm{M} :_{\kappa} \mathrm{S}$.
3. By induction, $\Gamma, \mathrm{X}{:}\mathrm{R}' \vdash \mathrm{M} \Rightarrow_{\kappa''} \mathrm{R}''$, where $\Gamma, \mathrm{X}{:}\mathrm{R}' \vdash \mathrm{R}'' \leq \mathrm{S}$ and $\kappa'' \sqsubseteq \kappa$.
4. Thus, $\Gamma \vdash \mathsf{let}\ \mathrm{X} = \mathrm{M}'\ \mathsf{in}\ (\mathrm{M} : \mathrm{S}) \Rightarrow_{\kappa' \sqcup \kappa''} \mathrm{S}$, and $\kappa' \sqcup \kappa'' \sqsubseteq \kappa$.

- Case: Rule 102. By induction, using Part 3.

- Case: Rules 103 and 104. Straightforward.

■

The soundness and completeness of the signature checking algorithm allow us to observe that sealed module expressions are truly *abstract* in the following sense. Suppose a well-formed program contains within it the module expression $\mathrm{M} :>_{\kappa} \mathrm{S}$. The algorithm regularizes the typing derivation of the program so that there is a unique subderivation of the well-formedness of $\mathrm{M} :>_{\kappa} \mathrm{S}$. The last rule applied in this subderivation is Rule 98, and it has conclusion $\Gamma \vdash (\mathrm{M} :>_{\kappa} \mathrm{S}) :_{\kappa \sqcup \kappa'} \mathrm{S}$ and premise $\Gamma \vdash \mathrm{M} :_{\kappa'} \mathrm{S}$, where Γ is some suitable context in which M is well-formed. Now, for any other implementation M' of S for which $\Gamma \vdash \mathrm{M}' :_{\kappa'} \mathrm{S}$, we can clearly swap $\mathrm{M}' :>_{\kappa} \mathrm{S}$ in for $\mathrm{M} :>_{\kappa} \mathrm{S}$ and the program will still be well-formed. Thus, there is no way for the part of the program outside of this sealed module expression to depend on the identities of any type components that are specified opaquely by S, as they may differ between M and M'.

Finally, since the signature checking algorithm, the context well-formedness judgment and all the signature judgments in Figure 4.5 are syntax-directed, it follows easily that the entire type system is decidable.

4.2.6 The Avoidance Problem

The type system for ML modules that I have presented in this chapter does not give a complete account of the ML module system, nor is it intended to. Rather, the goal of this type system is to capture what I believe are the most important and interesting aspects of the ML module system—sealing, functors and translucency—in an elegant type-theoretic framework. In addition, the type system illustrates that it is easy to support both total and partial functors, and both the basic and impure forms of sealing, within a single unified language design.

There are several features of ML that are difficult to account for directly in type theory but which will be dealt with formally in the language design of Part III using elaboration techniques. Most of these features are syntactic conveniences, whose practical importance should not be discounted but which are not very interesting from a type-theoretic point of view. As I discussed in Section 4.1.2, one particularly important one is ML's notion of signature matching, which is considerably more permissive than the signature subtyping judgment of this type system.

Another one, which is the subject of this section, is not so much a feature of ML as an issue that arises in a number of different guises and can be seen as affecting the typing rules for several different constructs in my type system. Recall that, for both second projections $\pi_2\mathrm{M}$ and functor applications $\mathrm{F}^\tau(\mathrm{M})$, the submodule M must be projectible in order for the whole module to be considered well-formed. One way to address this restriction is to support a variant of each of these constructs that permits the module M to be non-projectible, even impure, at the expense of requiring a signature annotation. In other words, consider extending the language with the constructs $(\pi_2\mathrm{M} : \mathrm{S})$ and $(\mathrm{F}^\tau(\mathrm{M}) : \mathrm{S})$, for both of which the principal signature will be S. These annotated constructs are in fact already expressible as derived forms:

$$\begin{array}{rcl} \pi_2(\mathrm{M}) : \mathrm{S} & \stackrel{\text{def}}{=} & \mathsf{let}\ \mathrm{X} = \mathrm{M}\ \mathsf{in}\ (\pi_2\mathrm{X} : \mathrm{S}) \\ \mathrm{F}^\tau(\mathrm{M}) : \mathrm{S} & \stackrel{\text{def}}{=} & \mathsf{let}\ \mathrm{X}_1 = \mathrm{F}\ \mathsf{in}\ \mathsf{let}\ \mathrm{X}_2 = \mathrm{M}\ \mathsf{in}\ (\mathrm{X_1}^\tau(\mathrm{X}_2) : \mathrm{S}) \end{array}$$

In practice, however, the signature annotation may constitute an unacceptable amount of syntactic overhead. We would like to have some way of inferring the signature S.

Unfortunately, it is not always possible to do so. As the above derived forms illustrate, the problem boils down to the desire for an *unannotated* let-module construct. Suppose that we had such a construct, written $\mathsf{let}\ \mathrm{X} = \mathrm{M}_1\ \mathsf{in}\ \mathrm{M}_2$, with exactly the same typing rule as annotated let's (Rule 101). Then clearly the above derived forms, minus the signature annotations, would give us a way of encoding second projections and functor applications in which the constituent module M need not be projectible.

The difficulty comes in computing the principal signature of $\mathsf{let}\ \mathrm{X} = \mathrm{M}_1\ \mathsf{in}\ \mathrm{M}_2$. Say that the principal signature of M_i is S_i. The signature S_2 may refer to the variable X. To construct a principal signature for the let, we need to find a minimal supersignature of S_2 that avoids reference to X. The "avoidance problem" [22, 45] is that such a minimal supersignature does not always exist. The same problem arises at the level of constructors and kinds as well: given a kind K that refers to a constructor variable α, there is not necessarily any minimal superkind of K that avoids α. For example, consider the kind $\mathrm{K} = (\mathbf{T} \to \mathfrak{S}(\alpha)) \times \mathfrak{S}(\alpha)$, which refers to a variable α bound with kind $\mathbf{T}$. One obvious superkind of K that avoids α is $(\mathbf{T} \to \mathbf{T}) \times \mathbf{T}$, but there are more precise ones. Specifically, for any type C of kind $\mathbf{T}$ that does not mention α, the kind $\Sigma\beta{:}(\mathbf{T} \to \mathbf{T}).\mathfrak{S}(\beta(\mathrm{C}))$ is an α-avoiding superkind of K. For different choices of C, however, the superkinds are incomparable, and there is no minimal one.

Going back to the original problem of allowing second projections and functor applications involving non-projectible modules, there is an alternative solution employed by Harper and Lillibridge [28] in their module type system. The idea is to allow $\pi_2\mathrm{M}$ and $\mathrm{F}^\tau(\mathrm{M})$, even when M is not projectible, so long as in the first case M can be given a non-dependent pair signature $\mathrm{S}_1 \times \mathrm{S}_2$, and in the second case F can be given a non-dependent functor signature $\mathrm{S}_1 \xrightarrow{\tau} \mathrm{S}_2$. In both cases, the signature of the result is merely S_2, avoiding any need to substitute a non-projectible module for a variable. Nevertheless, Harper and Lillibridge's type system still runs afoul of the avoidance problem. For instance, take the $\pi_2\mathrm{M}$ case. The principal signature of M may be a dependent pair signature $\mathrm{S} = \Sigma\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2$. Computing the principal signature of $\pi_2\mathrm{M}$ will thus require finding a minimal non-dependent supersignature of S. This is tantamount to finding a minimal supersignature of S_2 that avoids mentioning X^{c}, which is precisely the avoidance problem.

Different dialects of ML handle the avoidance problem—or do not handle it—in different ways. The type system of this chapter sidesteps the avoidance problem by requiring signature annotations on let-modules and restricting arguments to functors and second projections to be projectible.

```
# module type S = sig type t end
  module F = functor (X : S) ->
    struct type u = X.t type v = X.t end
  module G = functor (X : S) ->
    struct type u = X.t type v = u end
  module AppF = F((struct type t = int end : S))
  module AppG = G((struct type t = int end : S));;

(* Output of the Objective Caml 3.07+2 compiler *)
module type S = sig type t end
module F : functor (X : S) ->
  sig type u = X.t and v = X.t end
module G : functor (X : S) ->
  sig type u = X.t and v = u end
module AppF : sig type u and v end
module AppG : sig type u and v = u end
```

Figure 4.12: Encoding of the Avoidance Problem in O'Caml

Shao's type system makes similar restrictions, and these enable his language to support principal signatures [69].

On the other hand, both the Harper-Lillibridge "translucent sums" calculus [28] and Leroy's "manifest types" calculus [42] do not attempt to work around the avoidance problem at all, and thus lack principal signatures. Objective Caml, based on Leroy's work, also lacks principal signatures. Most of the time this does not cause serious problems, but on occasion it leads to unpredictable typechecking, as illustrated in the O'Caml code shown in Figure 4.12. Two functors `F` and `G` are defined that have equivalent, transparent principal signatures. Yet when the functors are applied to the same sealed module expression, the signatures of the results `AppF` and `AppG` differ rather arbitrarily, based on some purely syntactic discrepancy between the signatures of `F` and `G`.

The semantics of Standard ML, as described by Harper and Stone [32], addresses the avoidance problem differently, and in such a way that avoids the unpredictability of O'Caml typechecking. SML interprets the unannotated $\mathsf{let}\ \mathrm{X} = \mathrm{M}_1\ \mathsf{in}\ \mathrm{M}_2$ as if it were the *pair* module $\langle \mathrm{X} = \mathrm{M}_1, \mathrm{M}_2 \rangle$, and then ensures via elaboration techniques that the second component of this pair is the only one visible from outside the let. Applications of functors to non-projectible modules are handled by first rewriting them in terms of let-expressions (via the encoding given earlier in this section) and then interpreting the let's as pairs. (Second projections may be rewritten similarly, but SML chooses to only permit projections from paths.)

Do modules in SML have principal signatures? Yes and no. Under the Harper-Stone interpretation of SML, SML modules are translated to internal-language (IL) modules, and these IL modules do have principal signatures *in the IL*. However, the principal IL signature of an IL module does not necessarily correspond to any SML signature, so one cannot always write the principal signature of an SML module in SML itself. (As Shao would say, SML lacks "fully syntactic" signatures.) Returning to the example in Figure 4.12, if we were to write this code in SML, then `AppF` would receive the IL signature `sig type u = ?.t and v = ?.t end`, where `?.t` stands for the abstract `t` component of the unnamed argument module. There is no way to write this signature in SML itself, but at least we can count on the fact that `AppF.u` = `AppF.v`, which is not true in O'Caml.

There is a simple, reasonable tradeoff here: SML modules written without the use of unannotated let's (and the other features encoded in terms of them) will have principal signatures *in SML*, whereas modules that employ the more flexible let construct may not. The language I define in Part III follows the SML/Harper-Stone approach to dealing with the avoidance problem.

4.2.7 Module Phase-Splitting

In this final section, I define a phase-splitting translation for modules to match the one for signatures given in Section 4.1.4. The module translation explains how modules may be interpreted in terms of core-language constructs, avoiding the need to give a separate dynamic semantics and type safety proof for the module language itself. The translation does not, however, preserve all the data abstraction guarantees of the module language. In particular, it ignores basic sealing, in the sense that if M is a pure module then M and M $:>_{\mathsf{P}}$ S phase-split to the same result. I also define translations for terms and dynamic contexts that interpret the new term-level constructs and context binding forms introduced in this chapter in terms of existing core-language constructs.

The translations for modules, terms and contexts are shown in Figures 4.13 and 4.14. All of these translations produce syntactic objects that are well-formed in the type system of the core language of Chapter 3. I will refer to such syntactic objects as "core" objects.

The translation of dynamic contexts has the form $\Gamma \rightrightarrows \Gamma'$, mapping a module-language dynamic context Γ to a core context Γ'. For any core bindings of constructor variables α or value variables x, the translation is the identity. For a module variable binding X:S, the translation splits X into a constructor variable X^{c} (representing the static part of X) and a value variable X^{r} (representing the dynamic part of X).[10] The kind and type with which X^{c} and X^{r} are bound, respectively, are determined by phase-splitting X's signature S.

The translation of modules involves two judgments, one for pure modules and one for impure modules. These judgments are patterned on the signature synthesis algorithm of Figure 4.11, in the sense that the derivation of the translation of M, be it pure or impure, matches precisely the structure of the signature synthesis derivation for M. The difference between the pure and impure translation judgments is in their output. The pure judgment, written $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]$, splits M into a constructor C (representing the static part of M) and a term e (representing the dynamic part of M). It is modeled closely on the non-standard module equivalence rules set forth by Harper, Mitchell and Moggi [30] in their phase-distinction calculus. Impure modules, on the other hand, have the property that they cannot necessarily be phase-split in this way—the identities of their type components may not be knowable until run time. Therefore, the impure translation judgment has the form $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{I}} \mathrm{S} \rightrightarrows e$, where e is a core term with package type $\langle\!|\mathrm{S}|\!\rangle$.[11] This judgment does not really "split" M, it *packages* M as a term, so I call it an "impure packaging" judgment.

In a number of the translation rules—especially those for modules, like pairs, whose submodules may or may not be pure—it is very convenient to be able to implicitly coerce the result of pure module phase-splitting, which has the form $[\mathrm{C}, e]$, into the result of impure module packaging, which is just a term. This coercion is implemented by the judgment $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows e$, shown at the bottom of Figure 4.13. It merely takes the result $[\mathrm{C}, e]$ of phase-splitting M and pack's it with type $\langle\!|\mathrm{S}|\!\rangle$. This "pure packaging" judgment, together with the impure packaging judgment, defines a single packaging judgment of the form $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \mathrm{S} \rightrightarrows e$. This packaging judgment has the property that the output term e has package type $\langle\!|\mathrm{S}|\!\rangle$, where S is the principal signature of M.

[10] The use of X^{c} and X^{r} is due to Harper, Mitchell and Moggi [30]—the "c" in X^{c} stands for "compile-time," and the "r" in X^{r} stands for "run-time."

[11] Note that this package type $\langle\!|\mathrm{S}|\!\rangle$ is really a core type because $\langle\!|\mathrm{S}|\!\rangle$ is just a macro—the module language of this chapter has not extended the core language of types in any way.

Context phase-splitting: $\Gamma \rightrightarrows \Gamma'$

$$\frac{}{\emptyset \rightrightarrows \emptyset} \qquad \frac{\Gamma \rightrightarrows \Gamma'}{\Gamma, \alpha{:}\mathrm{K} \rightrightarrows \Gamma', \alpha{:}\mathrm{K}} \qquad \frac{\Gamma \rightrightarrows \Gamma'}{\Gamma, x{:}\mathrm{C} \rightrightarrows \Gamma', x{:}\mathrm{C}} \qquad \frac{\Gamma \rightrightarrows \Gamma' \quad \mathrm{S} \rightrightarrows \llbracket \mathrm{X^c}{:}\mathrm{K.C} \rrbracket}{\Gamma, \mathrm{X}{:}\mathrm{S} \rightrightarrows \Gamma', \mathrm{X^c}{:}\mathrm{K}, \mathrm{X^r}{:}\mathrm{C}}$$

Pure module phase-splitting: $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]$

$$\frac{\mathrm{X}{:}\mathrm{S} \in \Gamma}{\Gamma \vdash \mathrm{X} \Rightarrow_{\mathsf{P}} \mathfrak{s}_{\mathrm{S}}(\mathrm{X}) \rightrightarrows [\mathrm{X^c}, \mathrm{X^r}]} \qquad \frac{}{\Gamma \vdash \langle\rangle \Rightarrow_{\mathsf{P}} 1 \rightrightarrows [\langle\rangle, \langle\rangle]}$$

$$\frac{\Gamma \vdash \mathrm{C} \Rightarrow \mathrm{K}}{\Gamma \vdash [\mathrm{C}] \Rightarrow_{\mathsf{P}} \llbracket \mathrm{K} \rrbracket \rightrightarrows [\mathrm{C}, \langle\rangle]} \qquad \frac{\Gamma \vdash e \Rightarrow \mathrm{C}}{\Gamma \vdash [e] \Rightarrow_{\mathsf{P}} \llbracket \mathrm{C} \rrbracket \rightrightarrows [\langle\rangle, e]}$$

$$\frac{\Gamma \vdash \mathrm{M}_1 \Rightarrow_{\mathsf{P}} \mathrm{S}_1 \rightrightarrows [\mathrm{C}_1, e_1] \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{M}_2 \Rightarrow_{\mathsf{P}} \mathrm{S}_2 \rightrightarrows [\mathrm{C}_2, e_2]}{\Gamma \vdash \langle \mathrm{X} = \mathrm{M}_1, \mathrm{M}_2 \rangle \Rightarrow_{\mathsf{P}} \Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\langle \mathrm{X^c} = \mathrm{C}_1, \mathrm{C}_2 \rangle, \mathsf{let}\ \mathrm{X^c} = \mathrm{C}_1\ \mathsf{in}\ \mathsf{let}\ \mathrm{X^r} = e_1\ \mathsf{in}\ \langle \mathrm{X^r}, e_2 \rangle]}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\mathrm{C}, e]}{\Gamma \vdash \pi_1 \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S}_1 \rightrightarrows [\pi_1 \mathrm{C}, \pi_1 e]} \qquad \frac{\Gamma \vdash \mathbb{M} \Rightarrow_{\mathsf{P}} \Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\mathrm{C}, e]}{\Gamma \vdash \pi_2 \mathbb{M} \Rightarrow_{\mathsf{P}} \mathrm{S}_2[\pi_1 \mathbb{M}/\mathrm{X}] \rightrightarrows [\pi_2 \mathrm{C}, \pi_2 e]}$$

$$\frac{\Gamma \vdash \mathrm{S}_1 \rightrightarrows \llbracket \mathrm{X^c}{:}\mathrm{K}_1.\mathrm{C}_1 \rrbracket \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S}_2 \rightrightarrows [\mathrm{C}, e]}{\Gamma \vdash \lambda^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}_1.\mathrm{M} \Rightarrow_{\mathsf{P}} \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\lambda \mathrm{X^c}{:}\mathrm{K}_1.\mathrm{C},\ \Lambda \mathrm{X^c}{:}\mathrm{K}_1.\lambda \mathrm{X^r}{:}\mathrm{C}_1.e]}$$

$$\frac{\Gamma \vdash \mathrm{F} \Rightarrow_{\mathsf{P}} \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\mathrm{C}_1, e_1] \quad \Gamma \vdash \mathbb{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathrm{C}_2, e_2] \quad \Gamma \vdash \mathrm{S} \leq \mathrm{S}_1}{\Gamma \vdash \mathrm{F}^{\mathsf{tot}}(\mathbb{M}) \Rightarrow_{\mathsf{P}} \mathrm{S}_2[\mathbb{M}/\mathrm{X}] \rightrightarrows [\mathrm{C}_1(\mathrm{C}_2),\ e_1[\mathrm{C}_2](e_2)]}$$

$$\frac{\Gamma \vdash \mathrm{S}_1 \rightrightarrows \llbracket \mathrm{X^c}{:}\mathrm{K}_1.\mathrm{C}_1 \rrbracket \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{M} \Rightarrow_{\kappa} \mathrm{S} \rightrightarrows e \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{S} \leq \mathrm{S}_2}{\Gamma \vdash \lambda^{\mathsf{par}}(\mathrm{X}{:}\mathrm{S}_1){:}\mathrm{S}_2.\mathrm{M} \Rightarrow_{\mathsf{P}} \Pi^{\mathsf{par}} \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows [\langle\rangle,\ \Lambda \mathrm{X^c}{:}\mathrm{K}_1.\lambda \mathrm{X^r}{:}\mathrm{C}_1.\mathsf{coerce}\ e\ \mathsf{to}\ \llparenthesis \mathrm{S}_2 \rrparenthesis]}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S}' \rightrightarrows [\mathrm{C}, e] \quad \Gamma \vdash \mathrm{S}' \leq \mathrm{S}}{\Gamma \vdash \mathrm{M}\ {:}{>}_{\mathsf{P}}\ \mathrm{S} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \mathbb{S} \rightrightarrows e \quad \mathbb{S} \rightrightarrows \llbracket \alpha{:}\mathbb{K}.\mathrm{C} \rrbracket}{\Gamma \vdash \mathsf{purify}(\mathrm{M}) \Rightarrow_{\mathsf{P}} \mathbb{S} \rightrightarrows [\mathsf{Can}(\mathbb{K}),\ \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ e\ \mathsf{in}\ (x : \mathrm{C}[\mathsf{Can}(\mathbb{K})/\alpha])]}$$

$$\frac{\Gamma \vdash \mathrm{M}_1 \Rightarrow_{\mathsf{P}} \mathrm{S}_1 \rightrightarrows [\mathrm{C}_1, e_1] \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{M}_2 \Rightarrow_{\mathsf{P}} \mathrm{S}_2 \rightrightarrows [\mathrm{C}_2, e_2] \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{S}_2 \leq \mathrm{S} \quad \Gamma \vdash \mathrm{S}\ \mathrm{sig}}{\Gamma \vdash \mathsf{let}\ \mathrm{X} = \mathrm{M}_1\ \mathsf{in}\ (\mathrm{M}_2 : \mathrm{S}) \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathsf{let}\ \mathrm{X^c} = \mathrm{C}_1\ \mathsf{in}\ \mathrm{C}_2,\ \mathsf{let}\ \mathrm{X^c} = \mathrm{C}_1\ \mathsf{in}\ \mathsf{let}\ \mathrm{X^r} = e_1\ \mathsf{in}\ e_2]}$$

Pure module packaging: $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows e$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]}{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows \mathsf{pack}\ [\mathrm{C}, e]\ \mathsf{as}\ \llparenthesis \mathrm{S} \rrparenthesis}$$

Figure 4.13: Module, Term and Context Translation

Impure module packaging: $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{I}} \mathrm{S} \rightrightarrows e$

$$\frac{\Gamma \vdash \mathrm{M}_1 \Rightarrow_{\kappa_1} \mathrm{S}_1 \rightrightarrows e_1 \quad \Gamma, \mathrm{X}_1{:}\mathrm{S}_1 \vdash \mathrm{M}_2 \Rightarrow_{\kappa_2} \mathrm{S}_2 \rightrightarrows e_2 \quad \kappa_1 \sqcup \kappa_2 = \mathtt{I}}{\begin{array}{l}\Gamma \vdash \langle \mathrm{X}_1 = \mathrm{M}_1, \mathrm{M}_2 \rangle \Rightarrow_{\mathtt{I}} \Sigma \mathrm{X}_1{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows \\ \quad \mathsf{let}\ [\mathrm{X}_1^{\mathsf{c}}, \mathrm{X}_1^{\mathsf{r}}] = \mathsf{unpack}\ e_1\ \mathsf{in}\ \mathsf{let}\ [\mathrm{X}_2^{\mathsf{c}}, \mathrm{X}_2^{\mathsf{r}}] = \mathsf{unpack}\ e_2\ \mathsf{in}\ \mathsf{pack}\ [\langle \mathrm{X}_1^{\mathsf{c}}, \mathrm{X}_2^{\mathsf{c}} \rangle, \langle \mathrm{X}_1^{\mathsf{r}}, \mathrm{X}_2^{\mathsf{r}} \rangle]\ \mathsf{as}\ \langle\!| \Sigma \mathrm{X}_1{:}\mathrm{S}_1.\mathrm{S}_2 |\!\rangle\end{array}}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{I}} \Sigma \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows e}{\Gamma \vdash \pi_1 \mathrm{M} \Rightarrow_{\mathtt{I}} \mathrm{S}_1 \rightrightarrows \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ e\ \mathsf{in}\ \mathsf{pack}\ [\pi_1 \alpha, \pi_1 x]\ \mathsf{as}\ \langle\!| \mathrm{S}_1 |\!\rangle}$$

$$\frac{\Gamma \vdash \mathrm{F} \Rightarrow_{\mathtt{I}} \Pi^{\mathsf{tot}} \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows e \quad \Gamma \vdash \mathbb{M} \Rightarrow_{\mathtt{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e'] \quad \Gamma \vdash \mathrm{S} \leq \mathrm{S}_1}{\Gamma \vdash \mathrm{F}^{\mathsf{tot}}(\mathbb{M}) \Rightarrow_{\mathtt{I}} \mathrm{S}_2[\mathbb{M}/\mathrm{X}] \rightrightarrows \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ e\ \mathsf{in}\ \mathsf{pack}\ [\alpha(\mathrm{C}), x[\mathrm{C}](e')]\ \mathsf{as}\ \langle\!| \mathrm{S}_2[\mathbb{M}/\mathrm{X}] |\!\rangle}$$

$$\frac{\Gamma \vdash \mathrm{F} \Rightarrow_{\kappa} \Pi^{\mathsf{par}} \mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2 \rightrightarrows e \quad \Gamma \vdash \mathbb{M} \Rightarrow_{\mathtt{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e'] \quad \Gamma \vdash \mathrm{S} \leq \mathrm{S}_1}{\Gamma \vdash \mathrm{F}^{\mathsf{par}}(\mathbb{M}) \Rightarrow_{\mathtt{I}} \mathrm{S}_2[\mathbb{M}/\mathrm{X}] \rightrightarrows \mathsf{let}\ [\_\, , x] = \mathsf{unpack}\ e\ \mathsf{in}\ (x[\mathrm{C}](e') : \langle\!| \mathrm{S}_2[\mathbb{M}/\mathrm{X}] |\!\rangle)}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa'} \mathrm{S}' \rightrightarrows e \quad \Gamma \vdash \mathrm{S}' \leq \mathrm{S} \quad \kappa \sqcup \kappa' = \mathtt{I}}{\Gamma \vdash (\mathrm{M}\ {:}{>}_{\kappa}\ \mathrm{S}) \Rightarrow_{\mathtt{I}} \mathrm{S} \rightrightarrows \mathsf{coerce}\ e\ \mathsf{to}\ \langle\!| \mathrm{S} |\!\rangle}$$

$$\frac{\Gamma \vdash \mathrm{S}\ \mathrm{sig} \quad \Gamma \vdash e \Rightarrow \mathrm{C} \rightrightarrows e' \quad \Gamma \vdash \mathrm{C} \equiv \langle\!| \mathrm{S} |\!\rangle : \mathbf{T}}{\Gamma \vdash \mathsf{unpack}\ e\ \mathsf{as}\ \mathrm{S} \Rightarrow_{\mathtt{I}} \mathrm{S} \rightrightarrows e'}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{M}_1 \Rightarrow_{\kappa_1} \mathrm{S}_1 \rightrightarrows e_1 \quad \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{M}_2 \Rightarrow_{\kappa_2} \mathrm{S}_2 \rightrightarrows e_2 \\ \Gamma, \mathrm{X}{:}\mathrm{S}_1 \vdash \mathrm{S}_2 \leq \mathrm{S} \quad \Gamma \vdash \mathrm{S}\ \mathrm{sig} \quad \kappa_1 \sqcup \kappa_2 = \mathtt{I}\end{array}}{\Gamma \vdash \mathsf{let}\ \mathrm{X} = \mathrm{M}_1\ \mathsf{in}\ (\mathrm{M}_2 : \mathrm{S}) \Rightarrow_{\mathtt{I}} \mathrm{S} \rightrightarrows \mathsf{let}\ [\mathrm{X}^{\mathsf{c}}, \mathrm{X}^{\mathsf{r}}] = \mathsf{unpack}\ e_1\ \mathsf{in}\ \mathsf{coerce}\ e_2\ \mathsf{to}\ \langle\!| \mathrm{S} |\!\rangle}$$

Term translation: $\Gamma \vdash e \Rightarrow \mathrm{C} \rightrightarrows e'$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} [\![\mathrm{C}]\!] \rightrightarrows e}{\Gamma \vdash \mathsf{Term}(\mathrm{M}) \Rightarrow \mathrm{C} \rightrightarrows \mathsf{let}\ [\_\, , x] = \mathsf{unpack}\ e\ \mathsf{in}\ (x : \mathrm{C})}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \mathrm{S}' \rightrightarrows e \quad \Gamma \vdash \mathrm{S}' \leq \mathrm{S}}{\Gamma \vdash \mathsf{pack}\ \mathrm{M}\ \mathsf{as}\ \mathrm{S} \Rightarrow \langle\!| \mathrm{S} |\!\rangle \rightrightarrows \mathsf{coerce}\ e\ \mathsf{to}\ \langle\!| \mathrm{S} |\!\rangle}$$

All other term phase-splitting rules are defined in the obvious inductive manner, *e.g.*,

$$\frac{\Gamma \vdash v_1 \Rightarrow \mathrm{C}_1 \rightrightarrows v_1' \quad \Gamma \vdash v_2 \Rightarrow \mathrm{C}_2 \rightrightarrows v_2'}{\Gamma \vdash \langle v_1, v_2 \rangle \Rightarrow \mathrm{C}_1 \times \mathrm{C}_2 \rightrightarrows \langle v_1', v_2' \rangle}$$

Figure 4.14: Module, Term and Context Translation (continued)

The term translation judgment has the form $\Gamma \vdash e \Rightarrow \mathrm{C} \rightrightarrows e'$, where e' is guaranteed to be core and still have e's type C. For all the core term constructs, this term translation simply recurses on the subterms. The translation only does something for the term constructs involving modules.

Both the term and module translation judgments make frequent use of the following syntactic sugar for coercing from one package type to another:

$$\mathsf{coerce}\ e\ \mathsf{to}\ \langle\!|\mathrm{S}|\!\rangle \quad \stackrel{\mathrm{def}}{=} \quad \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ e\ \mathsf{in}\ \mathsf{pack}\ [\alpha, x]\ \mathsf{as}\ \langle\!|\mathrm{S}|\!\rangle$$

It is easy to check that the term $\mathsf{coerce}\ e\ \mathsf{to}\ \langle\!|\mathrm{S}|\!\rangle$ is well-formed (with type $\langle\!|\mathrm{S}|\!\rangle$) whenever e has type $\langle\!|\mathrm{S}'|\!\rangle$ and S′ is a subtype of S.

As for the rules themselves: I have already described how most of the pure module phase-splitting rules work in my discussion of signature phase-splitting in Section 4.1.4. The only one that requires special comment here is the rule for $\mathsf{purify}(\mathrm{M})$. In this case, the module M may be impure, so the premise requires only that M translate to the package e. Ordinarily this does not give us a way of phase-splitting M. However, since M's signature $\mathbb{S}$ is transparent, we know that there is a canonical constructor[12] $\mathsf{Can}(\mathsf{Fst}(\mathbb{S}))$ of kind $\mathsf{Fst}(\mathbb{S})$, which can be used to represent the static part of M. Then, to obtain M's dynamic part, we simply unpack e into α and x, and then export x. There is no problem with the constructor variable α escaping its scope because $\mathsf{Can}(\mathsf{Fst}(\mathbb{S}))$ is equivalent to α and can thus be substituted for α in the type of x.

The packaging rules for impure modules are not fundamentally that different from the phase-splitting rules for pure modules. The rule for pair modules, for instance, pairs the static parts of the two submodules into one constructor and pairs their dynamic parts into one term. The difference is that the static parts of the submodules are not statically known. All we have for each submodule M_i is its translation as the package e_i. We must therefore first unpack each e_i into $\mathrm{X}_i^{\mathsf{c}}$ and $\mathrm{X}_i^{\mathsf{r}}$ before we can pair the static parts together and the dynamic parts together. In the end, the result must be packaged up again as a term. Nearly all the packaging rules follow this same procedure: unpack, then phase-split, then pack.

Finally, here is a proposition summarizing soundness and related properties of the translations defined in this section. The proof goes through by straightforward induction.

Proposition 4.2.7 (Properties of Module, Term and Context Translation)
Assume $\Gamma \vdash \mathrm{ok}$. Then, $\Gamma \rightrightarrows \Gamma'$ and:

1. $\Gamma' \vdash \mathrm{ok}$ and Γ' is core.
2. $\mathsf{Fst}(\Gamma) = \mathsf{Fst}(\Gamma')$. Thus, for all static judgments $\mathcal{J}$, $\Gamma \vdash \mathcal{J}$ if and only if $\Gamma' \vdash \mathcal{J}$.
3. $\Gamma \vdash e \Rightarrow \mathrm{C}$ if and only if there exists a core e' such that $\Gamma \vdash e \Rightarrow \mathrm{C} \rightrightarrows e'$.
4. If $\Gamma \vdash e \Rightarrow \mathrm{C} \rightrightarrows e'$, then $\Gamma' \vdash e' : \mathrm{C}$. Also, if e is a value, then e' is a value.
5. $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \mathrm{S}$ if and only if there exists a core e such that $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \mathrm{S} \rightrightarrows e$.
6. If $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa} \mathrm{S} \rightrightarrows e$, then $\Gamma' \vdash e : \langle\!|\mathrm{S}|\!\rangle$.
7. $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S}$ if and only if there exist core C and e such that $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]$.
8. If $\Gamma \vdash \mathrm{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]$ and $\mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{D}]\!]$, then $\Gamma' \vdash \mathrm{C} : \mathrm{K}$ and $\Gamma' \vdash e : \mathrm{D}[\mathrm{C}/\alpha]$.
9. If $\Gamma \vdash \mathbb{M} \Rightarrow_{\mathsf{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]$, then $\mathrm{C} = \mathsf{Fst}(\mathbb{M})$.

Proof: By straightforward induction on the translation. ■

[12]Canonical constructors $\mathsf{Can}(\mathbb{K})$ were defined in Section 3.1.1.

Part II

Recursive Modules

Chapter 5

The Recursive Module Problem

Recursive modules are one of the most frequently requested extensions to the ML languages. After all, the ability to have cyclic dependencies between pieces of code written in separate files is a feature that is commonplace in mainstream languages like C and Java, languages which are not nearly as expressive as ML in other ways. From the programmer's perspective, it seems very strange that the ML module system should provide such powerful mechanisms for data abstraction and code reuse, and yet not provide any support for recursive modules. Certainly, for simple examples of recursive modules, it is difficult to convincingly argue why ML could not be extended to allow them. However, when one considers the semantics of a *general* recursive module mechanism, one runs into a host of interesting problems for which the "right" solutions are far from obvious.

In this chapter, I explore at a high level the problem of extending ML with recursive modules. I begin in Section 5.1 by giving several examples of how recursive modules would constitute a useful extension to ML. In Section 5.2, I lay out the key concepts and issues that arise in the design of a recursive module extension. Many of the ideas in this section are based on previous work by Crary, Harper and Puri [6], but there are a number of new observations as well. In Section 5.3, I examine a number of existing recursive module proposals, and weigh their advantages and disadvantages. Finally, in Section 5.4, I describe my own proposal for a recursive module extension to ML, which I will formalize fully in Part III.

While much of my proposed semantics for recursive modules relies on elaboration techniques, it also involves some extensions to my type system for modules from Chapters 3 and 4. These extensions are presented in Chapter 6. In addition, there is one aspect of recursive modules—namely, the static detection of "unsafe" (or "ill-founded") recursive definitions—that I discuss only briefly in the present chapter. This topic is examined more thoroughly in Chapter 7.

5.1 Motivating Examples

There are several reasons why recursive modules would be useful to have in ML. The primary one is to enhance the language's support for data abstraction. One of the main methodological goals of modules is to help the programmer weaken the dependencies between program components by breaking them into separate modules. However, if `f` and `g` are mutually recursive functions, then the ML programmer is forced to write their definitions together in the same module. This restriction is unfortunate because it means that there is no way to hide information about the implementation of `f` from the implementation of `g` or vice versa—they are typechecked in the same typing context. Similarly, if `t` and `u` are mutually recursive `datatype`'s, then the ML programmer is forced to define

```
structure rec Expr :> sig type t ; val eval : t -> t ; ... end =
struct
  datatype t = VarExpr of var | LetExpr of Bind.t * t | ...
  fun eval (e : t) : t =
     case e of ...
        | LetExpr (b,e) => ... Bind.eval(b) ...
  ...
end
and Bind :> sig type t ; val eval : t -> (var * Expr.t) list ; ... end =
struct
  datatype t = ... | ValBind of var * Expr.t | ...
  fun eval (b : t) : (var * Expr.t) list =
     case b of ...
        | ValBind (v,e) => [(v,Expr.eval(e))]
  ...
end
```

Figure 5.1: Mutually Recursive Modules `Expr` and `Bind`

them together as well. Forcing program components to be written together, regardless of whether they belong together conceptually, contradicts the ideals of data abstraction and code reuse.

Figure 5.1 shows a canonical example of where it might be desirable to break mutually recursive types and functions into separate modules. There are two modules, `Expr` and `Bind`. Each module provides a type `t` representing a different syntactic class—"expressions" in one case, "bindings" in the other—in the abstract syntax of some language, along with an `eval` function that implements the dynamic semantics of that language. For `Expr`, the `eval` function evaluates an expression (represented by its argument) to a value, and returns the piece of abstract syntax (of type `Expr.t`) corresponding to that value. For `Bind`, the `eval` function evaluates the binding(s) represented by its argument, and returns a list of variable-value pairs.

Note that the types `Expr.t` and `Bind.t` have mutually recursive definitions, as do the functions `Expr.eval` and `Bind.eval`. One of the benefits of breaking these types and functions into separate modules is that the implementation of `Expr` cannot rely on how `Bind.t` is defined, nor can `Bind` rely on how `Expr.t` is defined, because each module is sealed with an opaque interface that hides the definition of its `t` component. This ensures that the implementation of each module will remain well-typed under changes to the other module, so long as those changes do not affect its interface. Admittedly, for this example to make any sense, we do need to provide some extra functionality in these interfaces so that one may actually *create* a value of type `Expr.t` or `Bind.t`, but doing so does not require us to reveal the definitions of those types.

There are at least two common techniques that programmers use in practice to work around ML's restrictions and make up for the lack of a `structure rec` construct. Let's say we are trying to break up functions `f` and `g` into separate modules `A` and `B`. One technique, shown in Figure 5.2, is to first define a preliminary module `PreA` that defines `f` as parameterized over a definition for `g` since `g` has not been defined yet. The module `B` can then define `g`, with references to "`f`" replaced by `PreA.f(g)`. Finally, we can write the real module `A`, which "ties the recursive knot" by defining the real `f` as the instantiation of the preliminary `PreA.f` with the real `B.g`. One can similarly break up mutually recursive `datatype` definitions by parameterizing one type over the other and

```
structure PreA = struct
  fun f g x = ... g(...) ...
end
structure B = struct
  fun g y = let val f = PreA.f g in ... f(...) ...
end
structure A = struct
  val f = PreA.f B.g
end
```

Figure 5.2: Parameterization Workaround for Separating Recursive Function Definitions

```
structure A = struct
  local
    val ref_g : (C1 -> C2) ref = ref (fn _ => raise Error)
    fun g x = (!ref_g) x
  in
    fun install_g real_g = (ref_g := real_g)
    fun f x = ... g(...) ...
  end
end
structure B = ... (* defines g directly in terms of A.f *) ...
val _ = A.install_g B.g
```

Figure 5.3: Backpatching Workaround for Separating Recursive Function Definitions

instantiating it later on. This parameterization technique is rather awkward, though, as it requires the definition of a whole extra module (`PreA`).

Figure 5.3 illustrates an alternative workaround, which is similar to Scheme's "backpatching" semantics for recursive definitions [38]. The idea is to define in module `A` a reference cell `ref_g` containing a dummy function, which will eventually be updated (backpatched) with the real `B.g`. Uses of `g` inside the body of `A.f` will first dereference `ref_g` to obtain the real `g` and then apply the result. The module `B` can then define `g` directly in terms of `A.f`. Once `B` is defined, `A`'s `ref_g` can be backpatched via the `install_g` function that `A` provides. Despite its use of mutable state, this approach is somewhat cleaner than the parameterization workaround because it does not require one to define two versions of the module `A`. In addition, it localizes the dirty work to module `A`; module `B` is written as if its reference to `A.f` were strictly hierarchical. On the other hand, it is limited in that it does not help in separating mutually recursive `datatype`'s.

The above techniques are essentially stopgap solutions—they work acceptably on a small scale, but are no substitute for a real recursive module mechanism. Moreover, there are examples that are only expressible in the presence of recursive modules. The best-known such example is an implementation of "bootstrapped heaps" from Okasaki's thesis [59]. A stripped-down version of this example (adapted from Russo [66]) is shown in Figure 5.4.

The `Bootstrap` functor takes as input a functor `MkHeap` providing an implementation of heaps, and a module `Ord` providing some type of ordered elements. It returns a module implementing heaps

```
signature ORDERED = sig type t ; val leq : t * t -> bool end
signature HEAP = sig
  structure Elem : ORDERED
  type t
  val empty : t
  val insert : Elem.t * t -> t
  val merge : t * t -> t
  val findMin : t -> Elem.t option
end

functor Bootstrap (functor MkHeap : (X : ORDERED) ->
                                      HEAP where type Elem.t = X.t
                   structure Ord : ORDERED)
           :> HEAP where type Elem.t = Ord.t =
struct
  structure Elem = Ord
  structure rec Boot :>
    sig
        datatype t = E | H of Elem.t * Heap.t
        val leq : t * t -> bool
    end =
  struct
    datatype t = E | H of Elem.t * Heap.t
    fun leq (H(x,_), H(y,_)) = Elem.leq(x,y)
  end
  and Heap :> HEAP where type Elem.t = Boot.t =
    MkHeap(Boot)

  open Boot     (* type t = Boot.t *)
  val empty = E
  fun merge (E,h) = h
    | merge (h,E) = h
    | merge (h1 as H(x,p1), h2 as H(y,p2)) =
      if Elem.leq(x,y) then H(x,Heap.insert(h2,p1)
      else H(y,Heap.insert(h1,p2)
  fun insert (x,h) = merge (H(x,Heap.empty),h)
  fun findMin E = NONE
    | findMin (H(x,_)) = SOME x
end
```

Figure 5.4: Bootstrapped Heap Example

```
datatype 'a t = A | B of 'a * 'a t t

structure rec X :> sig val collect : 'a t -> 'a list end =
struct
  fun collect (A) = []
    | collect (B(x,TT)) =       (* x : C, TT : C * C t t *)
    let val TL = X.collect TT   (* X.collect : C t t -> C t list *)
        val LL = map collect TL (* collect : C t -> C list *)
        val L = flatten LL      (* flatten : C list list -> C list *)
    in x::L end
end
```

Figure 5.5: Encoding Polymorphic Recursion Using a Recursive Module

of `Ord.t`'s, for which the `merge` and `findMin` operations are constant-time, assuming that those operations were $O(\log n)$ and that `insert` was constant-time in `MkHeap`'s original implementation.

A "bootstrapped" heap, *i.e.,* a value of this new heap type, is either the empty heap, `E`, or a node `H(x,h)`, where `x` is the minimum element in the heap, and `h` is a heap (implemented by `MkHeap`) containing the rest of the elements. However, this internal heap `h` is not a heap of `Ord.t`'s, it is a heap of bootstrapped heaps! In other words, the type of bootstrapped heaps is defined in terms of heaps of itself. There is no way to define such a type in existing dialects of ML, because there is no way to write a `datatype` definition that is recursive with a functor application. This is completely straightforward, though, in the presence of recursive modules.

It is worth noting that, in the type system for modules I presented in Chapters 3 and 4, the bootstrapped heap example *can* be encoded without the use of module-level recursion, thanks to the concept of phase separation. Assuming that `MkHeap` is a total/applicative functor, the type `Boot.t` could be defined recursively as follows:

$$\texttt{datatype t = E | H of Elem.t * MkHeap}^{\texttt{c}}\texttt{(t).t}$$

Recall that $\texttt{MkHeap}^{\texttt{c}}$ is the type constructor representing the static part of the `MkHeap` functor. It only takes the *type* of elements as its argument, not the comparison function, and the `t` component it returns describes heaps of those elements. By employing this definition for the type of bootstrapped heaps, we can break the dependency of the `Boot` structure on the `Heap` structure. Nevertheless, one can easily imagine a modified version of this example in which the dynamic part of `Boot` (*i.e.,* `Boot.leq`) referred recursively to the `Heap` structure as well. In that case, phase separation would not suffice to break the cyclic dependency.

Another example of how recursive modules fundamentally increase the language's expressiveness is that they enable polymorphic recursion, *i.e.,* the ability to write recursive polymorphic functions that instantiate their type arguments differently at different recursive calls. While ML allows polymorphic `datatype`'s to be "non-uniform"—meaning that their definitions may contain instantiations of themselves at different types—there is no way to write functions that recur over non-uniform datatypes because ML does not support polymorphic recursion. (Type inference is undecidable in the presence of full polymorphic recursion [34, 39].)

With recursive modules, however, polymorphic recursion comes for free. Figure 5.5 shows an illustrative, if somewhat contrived, example of a type `'a t` whose definition is similar to that of `'a list`, except that the tail of an `'a t` has type `'a t t` instead of `'a t`. The recursive structure

`X` defines a function `collect` that collects all the values of type `'a` stored within an `'a t` and forms an `'a list` of them. Within the definition of `collect`, the type variable `'a` is fixed to some particular unknown type—call it C—and the variable `collect` has the *monomorphic* type C `t -> ` C `list`. Yet, in the intermediate step defining the variable `TL`, we need `collect` to have type C `t t -> ` C `t list`. Instead we can make use of `X.collect`, which has the *polymorphic* type ∀`'a. 'a t -> 'a list`. By implicitly instantiating the type argument of `X.collect` with C `t`, we obtain a value of the desired type.

What the recursive module is essentially doing for us in this example is giving us a place to write down the type of `X.collect`. Polymorphic recursion could just as easily be added directly to the core ML language, if polymorphically recursive functions were required to be annotated with their types. Nonetheless, recursive modules seem to be useful in a number of situations, and they provide an elegant way of encoding various paradigms that are either awkward or impossible to implement in existing dialects of ML.

5.2 Key Issues in the Design of a Recursive Module Extension

In the motivating examples of the previous section, I made use of a make-believe "`structure rec`" binding of the form

$$\texttt{structure rec } X_1 \texttt{ : } S_1 \texttt{ = } M_1 \texttt{ and ... and } X_n \texttt{ : } S_n \texttt{ = } M_n$$

but I was deliberately vague about both the static and dynamic semantics of this binding form, relying instead on the reader's intuition. In this section we will delve deeper into the semantics of recursive modules and discover that it is in fact quite difficult to explain formally what the "right" semantics should be. Along the way, I will introduce the reader to a number of key concepts and issues that arise in the design of a recursive module extension. These will serve as helpful guides in the discussion of the existing recursive module proposals in Section 5.3 and of my own recursive module proposal in Section 5.4.

5.2.1 Dynamic Semantics

Consider extending the module system of Chapter 4 with a new recursive construct $\mathsf{rec}(X : S.\, M)$.[1] The idea is that X is the module variable by which the module M refers to itself recursively. The signature S is needed for two reasons: (1) to know what signature to assign to X while typechecking M, and (2) to have a signature to assign to the recursive module itself, since the principal signature of M may very well refer to the bound variable X. (Note: this implies that S should be well-formed outside the recursive module and therefore not refer to X.) Some terminology: I will refer to X as the *recursive (module) variable*, S as the *declared signature*, and M as the *recursive module body*.

Let us begin by focusing on the dynamic semantics of this recursive construct. Formally speaking, in the way I have organized my module type system, modules are evaluated by first phase-splitting them and then evaluating the results according to the core-language dynamic semantics. However, that is primarily a technical device, allowing me to describe module compilation and prove type safety in one fell swoop. There should always be some intuitive way of explaining how modules are evaluated.

So, intuitively, how should $\mathsf{rec}(X : S.\, M)$ be evaluated? Under the standard interpretation of recursion via a fixed-point operator, $\mathsf{rec}(X : S.\, M)$ should evaluate to its "unrolling" $M[\mathsf{rec}(X : S.\, M)/X]$.

[1] While this construct only defines a single recursive module, mutually recursive modules are definable by writing them as substructures of a single recursive module. Examples of this are shown below.

```
rec (X : SIG.
struct
  structure A = struct
    val debug = ref false
    fun f(x) = ...X.B.g(x-1)...
  end
  structure B = struct
    val trace = ref false
    fun g(x) = ...X.A.f(x-1)...
  end
end)
```

Figure 5.6: Example of Recursive Module with Effects

Such a fixed-point semantics has the property that M is effectively re-evaluated at every recursive reference to X.

There is nothing inherently wrong with this behavior, and it will work fine for some purely functional recursive modules, such as the examples I gave in Section 5.1. It is undesirable, though, for recursive modules that contain computational effects. For example, consider the definition of two mutually recursive structures `A` and `B` shown in Figure 5.6. Here, `debug` and `trace` are externally-accessible debugging flags used by `f` and `g`, respectively. Under a fixed-point semantics for recursive modules, every recursive reference between `f` and `g` prompts a re-evaluation of the entire module, including the creation of brand new ref cells for `debug` and `trace`. In other words, each recursive call operates in an entirely different mutable state, so setting `debug` to `true` externally would not alter the fact that `!debug` is `false` during all recursive calls to `X.A.f` and `X.B.g`.

An alternative semantics for recursion that exhibits more appropriate behavior with respect to computational effects is the *backpatching* semantics used by Scheme [38], in which $\mathsf{rec}(\mathrm{X}\,{:}\,\mathrm{S}.\,\mathrm{M})$ would evaluate as follows: First, X is bound to a fresh location containing an undefined value; then, M is evaluated to a module *value* V; finally, X is backpatched with V. If the evaluation of M attempts to access the value of X, an error is reported. I described backpatching in Section 5.1 as one of the workarounds that programmers use in the absence of recursive modules, but here it is employed "behind the scenes," not explicitly by the programmer. Unlike the fixed-point semantics, backpatching has the advantage that it ensures the effects in M will only happen once.

Under a backpatching semantics, the question arises: How should we ensure that the recursion is "safe," *i.e.,* that the evaluation of M will not attempt to use the recursive variable X? Should we try to determine it *statically*, reporting an error at compile time if we cannot, or should we just insert a *dynamic* check at each use of X to make sure it has been backpatched? There is a common tradeoff here. The static approach allows for more efficient implementation, and compile-time detection is certainly preferable. On the other hand, any static method for ensuring safe recursion is bound to be conservative and may rule out perfectly safe recursive modules. For the time being, let us assume dynamic detection. I will explore static detection further in Chapter 7.

5.2.2 Recursively Dependent Signatures

Recall the example of the `Expr` and `Bind` modules from Figure 5.1. These mutually recursive modules may be encoded in terms of $\mathsf{rec}(\mathrm{X}\,{:}\,\mathrm{S}.\,\mathrm{M})$ by having M be a pair module whose first

```
sig
  structure Expr : sig
    type t
    (* makeLetExpr(b,e) constructs "let b in e" *)
    val makeLetExpr : Bind.t * t -> t
    (* matchLetExpr(e) returns SOME(b,e1) if e is of the form "let b in e1" *)
    val matchLetExpr : t -> (Bind.t * t) option
    ...
    val eval : t -> t
  end
  structure Bind : sig
    type t
    (* makeValBind(v,e) constructs "val v = e" *)
    val makeValBind : var * Expr.t -> t
    (* matchValBind(b) returns SOME(v,e) if b is of the form "val v = e" *)
    val matchValBind : t -> (var * Expr.t) option
    ...
    val eval : t -> (var * Expr.t) list
  end
end
```

Figure 5.7: Problematic Signature for `Expr` and `Bind`

component is `Expr` and whose second component is `Bind`. Joined into a single module, `Expr` and `Bind` can refer recursively to each other by projecting from the recursive variable X; that is, `Expr` can refer to `Bind` as X`.Bind`, and `Bind` can refer to `Expr` as X`.Expr`. In this case, `Bind` may also just refer to `Expr` directly, since `Expr` is defined first.

This encoding handles mutually recursive dependencies between the modules, but what do we do about recursive dependencies in the *signatures*? In other words, what do we do if the signature of each module refers to type components defined in the other module? These signatures, which are presumably part of the declared signature S, cannot contain references to the recursive variable X because S needs to be well-formed outside of the recursive module.

For example, suppose that, in addition to the `eval` function, the `Expr` module were to provide functions `makeLetExpr` and `matchLetExpr` for constructing and deconstructing `let` expressions of type `Expr.t`; and that the `Bind` module were similarly to provide functions `makeValBind` and `matchValBind` for constructing and deconstructing `val` bindings of type `Bind.t`. Figure 5.7 shows the resulting signature of the recursive module defining `Expr` and `Bind`. Note that the types of several function components of each submodule refer to the `t` component of the other submodule. The boxed references to `Bind.t` in the signature of `Expr` are not well-formed, since `Bind` is specified *after* `Expr` in the signature.

This problem was pointed out first by Crary *et al.* [6], who propose as a solution the introduction of a new signature form called a "recursively dependent signature" (or "rds"), written $\rho X.S$. The idea is that the bound variable X in $\rho X.S$ is a stand-in for the module that the signature is describing. In other words, a module M belongs to $\rho X.S$ precisely when it belongs to S[M/X]. Note that this definition only applies when M is projectible, for otherwise the substitution of $\mathsf{Fst}(M)$ for X^{c} in S is not valid. If M is not projectible, we can coerce it to an rds $\rho X.S$ by first let-binding it

```
ρX. sig
  structure Expr : sig
    type t
    val makeLetExpr : X.Bind.t * t -> t
    val matchLetExpr : t -> (X.Bind.t * t) option
    ...
  end
  structure Bind : sig
    type t
    ...
  end
end
```

Figure 5.8: Recursively Dependent Signature for `Expr` and `Bind`

to a variable Y, which *is* projectible. That is, we can expand M into let Y = M in (Y : ρX.S).

Figure 5.8 shows a recursively dependent signature that can be used as the declared signature for the `Expr` and `Bind` modules. The rds provides a way for the signature of `Expr` to refer to `Bind.t`, namely by projecting it from the bound variable `X`.

Recursively dependent signatures clearly add some power to ML's signature language, since the signature of `Expr` and `Bind` is not expressible without them. The question is: do rds's merely add flexibility to the *signature* language, or does the type structure of ML's *core* language need to be enhanced in some way in order to be able to support them, *i.e.,* to phase-split them? The answer to this question depends, perhaps unsurprisingly, on whether we place any restrictions on the kinds of recursive references to X that are permitted to occur in ρX.S.

Suppose that we place no restrictions, and consider the following rds:

```
ρX. sig type t = int * X.t end
```

A module M will inhabit this rds precisely when it also inhabits `sig type t = int *` M`.t end`, in which case we will be able to observe that M`.t` is equivalent to `int *` M`.t`. The only known way to account for this sort of recursive type equivalence is through the use of so-called "equi-recursive" type constructors [6]. An equi-recursive type constructor $\mu\alpha$:K.C has the property that it is the unique fixed-point of the constructor function $\lambda\alpha$:K.C, assuming one exists. In other words, $\mu\alpha$:K.C is equivalent to C[$\mu\alpha$:K.C/α], and if D is equivalent to C[D/α], then D is equivalent to $\mu\alpha$:K.C. Using equi-recursive types, the rds displayed above could be encoded as $[\![\mathfrak{S}(\mu\alpha{:}\mathbf{T}.\mathtt{int}\times\alpha)]\!]$.

Unfortunately, equi-recursive type constructors complicate the decision procedure for constructor equivalence [2], and their interaction with singleton kinds is not well understood. Furthermore, although the underlying type structure of ML must support *some* form of recursive type constructor in order to interpret recursive `datatype` definitions, equi-recursive types provide more type equivalences than necessary for this purpose. A `datatype` definition such as "`datatype t = Cons of int * t`" creates a new abstract type `t`, and the coercions between `t` and `int * t` are always witnessed at the programming level by an application of, or pattern match against, `Cons`. It is therefore not important to be able to observe that `t` and `int * t` are equivalent.

For the purpose of supporting ML's recursive types, "iso-recursive" type constructors are sufficient. Under an iso-recursive semantics, a type like $\mu\alpha{:}\mathbf{T}.\mathtt{int}\times\alpha$ is not equivalent to its unfolding,

```
ρX. sig
  structure Expr : sig
    datatype t = VarExpr of var | LetExpr of X.Bind.t * t | ...
    ...
  end
  structure Bind : sig
    datatype t = ... | ValBind of var * Expr.t | ...
    ...
  end
end
```

Figure 5.9: Rds for `Expr` and `Bind` with Mutually Recursive Datatype Specifications

`int` $\times$ ($\mu\alpha$:**T**.`int` $\times$ α), but the two types are isomorphic in the sense that values of either type can be coerced into the other type by a "fold" or "unfold" operation.[2] Iso-recursive type constructors have the advantage that they rely on a very simple equational theory, which is easy to incorporate into a core language with singleton kinds. They are not capable, however, of supporting the equational reasoning implied by unrestricted rds's like the one shown above. If we want to avoid the need for equi-recursive types, we need to impose some restrictions on rds's.

A simple restriction to ρX.S that was suggested by Crary *et al.* [6] as a way of avoiding equi-recursive types is to only permit references to X in S if they occur within the types of value components—references to X in the (singleton) kinds of type components are disallowed. This prevents one from writing the unrestricted rds shown above, since that rds contains a reference to X in the transparent specification—*i.e.,* the singleton kind—of the `t` component.

Viewed in the terms of my module type system, this restriction has the effect that $\mathrm{X^c}$ cannot appear in the free variables of Fst(S). As a result, it is possible to phase-split such rds's into the core language without requiring *any* recursive types. In particular, here is the phase-splitting rule:

$$\frac{\mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}]\!]}{\rho\mathrm{X}.\mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}[\alpha/\mathrm{X^c}]]\!]}$$

To understand why this rule makes sense, observe that the restriction described above requires all references to $\mathrm{X^c}$ in S to be *from* types appearing in C (the dynamic part of S) *to* types specified by K (the static part of S). (Crary *et al.* refer to this restriction as a "dynamic-on-static" restriction.) Thus, when we phase-split S, and the static and dynamic parts of S are separated, what were once "recursive" references to $\mathrm{X^c}$ in C can be replaced by direct references to α.

How prohibitive is the dynamic-on-static restriction? The original rds for `Expr` and `Bind` shown in Figure 5.8 is well-formed under it, but what about the rds shown in Figure 5.9, in which the `datatype` definitions of `Expr.t` and `Bind.t` are exposed? Since this signature contains a recursive reference to `X` in the specification of the *type* `Expr.t`, it would appear to be disallowed under the dynamic-on-static restriction.

Fortunately, though, it is not disallowed, because a `datatype` specification is not the same as a transparent type specification. Rather, as discussed above, `datatype`'s are *abstract*—an ML `datatype` specification for `t` may be viewed as an opaque type specification `type t`, together with a

[2]At higher kind, the semantics of iso-recursive type constructors becomes more difficult to explain, but I will do so precisely in Chapter 6.

sequence of value specifications for each of the constructors—*i.e.,* the branches—of the `datatype`.[3] The `datatype` specification for `Expr.t` implicitly introduces a value specification for each of its data constructors, including one for the function `LetExpr`, which has type `X.Bind.t * t -> t`. It is in the specification of `LetExpr` (and the other data constructors) that the recursive references to `X` occur, not in the specification of the type component `t` itself. Thus, we see that "`datatype`-on-static" recursive dependencies are really just a special case of dynamic-on-static dependencies. This makes dynamic-on-static rds's flexible enough to encode the signatures for all the motivating examples from Section 5.1.

Nevertheless, there are rds's that are understandable in terms of the existing ML core language—*i.e.,* without requiring the addition of equi-recursive types—but which the dynamic-on-static restriction does not permit. For instance, consider the following rds:

```
ρX. sig
  structure A : sig type t ; type u = X.B.u ; ... end
  structure B : sig type t ; type u = A.t * t ; ... end
end
```

While the definition of `A.u` refers to `X`, it does not introduce any truly cyclic dependencies at the level of individual type components. Specifically, `A.u` depends on `B.u`, but `B.u` only depends on the opaque types `A.t` and `B.t`, not on `A.u`. Correspondingly, some recursive module proposals loosen the dynamic-on-static restriction to permit recursive dependencies in transparent type specifications as well, so long as they are fundamentally acyclic.

On the other hand, the vanilla dynamic-on-static restriction has the benefit that it is very simple to explain type-theoretically. In particular, the well-formedness rule for dynamic-on-static rds's can be written as follows:

$$\frac{\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}) \vdash \mathrm{S}\ \mathrm{sig}}{\Delta \vdash \rho\mathrm{X.S}\ \mathrm{sig}}$$

Note that the premise of this rule implies that the context $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S})$ is well-formed. This in turn implies that $\mathsf{Fst}(\mathrm{S})$ is well-formed in the ambient context Δ, which is the essence of the dynamic-on-static restriction.

5.2.3 The Double Vision Problem

Having considered the well-formedness of recursively dependent signatures, let us now consider the well-formedness of recursive modules. A natural typing rule for $\mathsf{rec}(\mathrm{X}\,{:}\,\mathrm{S}.\,\mathrm{M})$ would be

$$\frac{\Gamma, \mathrm{X} \uparrow \mathrm{S} \vdash \mathrm{M} :_{\kappa} \mathrm{S}}{\Gamma \vdash \mathsf{rec}(\mathrm{X}\,{:}\,\mathrm{S}.\,\mathrm{M}) :_{\kappa} \mathrm{S}}$$

which checks that the recursive module body matches its declared signature. The new context binding form $\mathrm{X} \uparrow \mathrm{S}$ is needed because recursive variables are compiled differently from ordinary variables. In particular, X is represented not as a value of signature S, but as a memory location that will eventually be backpatched with a value of signature S. Correspondingly, references to X in M are not values either—they are computations that must implicitly check whether X has been backpatched and, if so, return its contents.

[3]In addition, a `datatype` specification implicitly specifies a function that enables a value of type `t` to be pattern-matched. The interpretation of ML `datatype` specifications presented here, which is based on Harper and Stone's treatment of SML semantics, will be formalized explicitly in the language definition of Chapter 9.

```
rec (X : EXPR_BIND.
let structure Body =
      struct
        structure Expr :> sig type t ... end = struct
          datatype t = VarExpr of var | LetExpr of X.Bind.t * t | ...
          ...
          fun makeLetExpr (b : X.Bind.t, e : t) : t = LetExpr(b,e)
          ...
          fun eval (e : t) : t =
           case e of ...
              | LetExpr (b,e) =>
                  let val (vs,es) = List.unzip (X.Bind.eval b)
                      (* vs : var list, es : X.Expr.t list *)
                  in  ...
                  end
          ...
        end
        structure Bind :> sig type t ... end = ...
      end
in  Body :> EXPR_BIND
end)
```

Figure 5.10: The Double Vision Problem Arising in `Expr` and `Bind`

Unfortunately, this rule is too simple—it fails to accept many recursive modules whose declared signatures contain *opaque* type specifications, including both the `ExprBind` example and the bootstrapped heap example from Section 5.1. Consider the code excerpt from the recursive module defining `Expr` and `Bind`, shown in Figure 5.10. In this version, the recursive module body is `let`-bound to a variable `Body`, which is then sealed with the declared rds `EXPR_BIND`. I have written the example this way primarily so that we have explicit names for the types defined in the body, *e.g.,* `Body.Expr.t` and `Body.Bind.t`. The declared signature `EXPR_BIND` is meant to stand for either one of the rds's given in Figures 5.8 and 5.9. Whichever rds is used, the type specifications in it are opaque. Consequently, when typechecking the recursive module body, there is no way to connect the abstract type components of the recursive variable `X`—namely, `X.Expr.t` and `X.Bind.t`—with the corresponding type components of `Body`. This results in several serious typing difficulties.

In order to understand these typing difficulties, let us first clarify what the typing rule actually requires. Assuming that the rds `EXPR_BIND` has the form ρX.S, the typing rule says that `Body` must match ρX.S, which means in turn that `Body` must match the signature S[`Body`/`X`].

The first problem is that the `Body.Expr.makeLetExpr` function does not match its required type. This function constructs a `let` expression of type `Body.Expr.t` from a binding of type `X.Bind.t` and an expression of type `Body.Expr.t`. However, in order for `Body` to match S[`Body`/`X`], the first argument of `Body.Expr.makeLetExpr` must have type `Body.Bind.t`, not `X.Bind.t`. There is no simple way of addressing this problem—at the point where `makeLetExpr` is defined, the type `Body.Bind.t` does not even exist yet! One might suggest switching the order of `Expr` and `Bind`, so that `Expr` can refer to `Bind` directly, but then the same problem would rear its head again when matching `Body.Bind.makeValBind` against *its* required type.

A second, more subtle problem arises in typechecking the body of the `Expr.eval` function. When the input to this function is of the form `LetExpr(b,e)`, the function makes a recursive call to `X.Bind.eval` in order to process the binding `b`. The return type of `X.Bind.eval` is `(var * X.Expr.t) list`; this list is then unzipped into a list (`vs`) of the variables bound by `b`, and a list (`es`) of the value expressions to which they are bound. That the expressions in `es` have type `X.Expr.t` is problematic for several reasons.

For one, the implementation of `eval` may want to deconstruct these expressions. Since they have type `X.Expr.t`, however, the only way to deconstruct them is to call the `match` functions provided by `X.Expr`, which is not as efficient or convenient as case-analyzing a value of the `datatype t` directly. Moreover, this problem *forces* the `Expr` module to provide deconstructor functions (or else expose the `datatype` definition of `t`) in its interface, regardless of whether it otherwise needs to.

In addition, suppose that the body `e` of the input expression `LetExpr(b,e)` has the form `VarExpr(v)`, where `v` is one of the variables bound in `b`. Presumably, in this case, the `eval` function should return the value expression in `es` that corresponds to `v`. Yet, the type of that expression will be `X.Expr.t`, whereas the required return type of `eval` is `Body.Expr.t`. The interface of `X` does not provide any way to coerce a value from `X.Expr.t` to `Body.Expr.t` or vice versa.

All of these typing difficulties are symptoms of what I call the "double vision problem." This problem is easiest to understand by thinking of `Expr` and `Bind` as being written, respectively, by two "agents" (or "principals") Alice and Bob [24]. Alice knows that the type `Expr.t` is implemented by the `datatype` definition shown in Figure 5.10, but Bob does not know this because it is not revealed in the interface for `Expr` that Alice provides. Similarly, Bob knows how `Bind.t` is implemented, but Alice does not.

In order to allow recursive references between the two modules, we have introduced the recursive variable `X`. Intuitively, since `X` will ultimately be backpatched with the result of evaluating the body, what either agent knows about the type components of `X` should *coincide* with what she knows about the definitions of the corresponding type components in the body. Thus, Alice should be allowed to know that `X.Expr.t` is implemented in the same way that she has implemented `Expr.t` in the body, and Bob should be allowed to know that `X.Bind.t` is implemented in the same way that he has implemented `Bind.t` in the body. Neither agent, however, should be allowed to know how the other agent's submodule of `X` is implemented. In addition, since Bob can refer to `Expr.t` in two ways—either directly or through `X`—he should be able to observe that the two types `Expr.t` and `X.Expr.t` coincide, without knowing what their underlying definition is.

The double vision problem is that the simple typing rule given at the beginning of this section fails to realize this intuition. Alice and Bob are shown the same interface for the recursive variable. If Alice wants to hide the identity of `X.Expr.t` from Bob, she must also hide it from herself. As a result, she "sees double"—that is, she sees `X.Expr.t` as being distinct from her own definition of `Expr.t`, and she cannot tell that they are really one and the same type. Bob also sees two versions of `Expr.t`, although he is not privy to a definition for either type.

One way to keep the simple typing rule and avoid double vision is to require that the declared signature of the recursive module be transparent. For example, if `X.Expr.t` were revealed in the declared signature to equal some type C, then Alice would be able to observe that `X.Expr.t` coincides with its implementation C. Of course, Bob would be able to observe this as well. Transparency addresses the double vision problem, but at the expense of preventing `Expr` and `Bind` from hiding type information from each other. Furthermore, this solution assumes that the programmer is *able* to write a transparent declared signature. In the case of `Expr` and `Bind`, the best that we can do is modify `EXPR_BIND` so that it exposes the `datatype` definitions of `Expr.t` and `Bind.t`. According to ML semantics, though, `datatype` specifications are opaque.

```
functor F_Expr (X : EXPR_BIND) = ...
...
functor F_Bind (X : EXPR_BIND) = ...
...
structure Link = rec (X : EXPR_BIND.
struct
  structure Expr = F_Expr(X)
  structure Bind = F_Bind(X)
end
```

Figure 5.11: Attempted Separate Compilation of `Expr` and `Bind`

The double vision problem is one of the most serious hurdles to overcome in designing a recursive module extension. In Section 5.3, I will discuss the conservative ways in which the existing recursive module proposals deal with it. A key contribution of my own recursive module design, which I describe in Section 5.4, is to provide a more general and effective cure for double vision.

5.2.4 Separate Compilation

One of the main motivations for recursive modules is to allow mutually recursive functions and data types to be broken into separate, mutually recursive modules. The recursive module construct $\mathsf{rec}(\mathrm{X}:\mathrm{S}.\,\mathrm{M})$ makes it possible to write mutually recursive modules, but the modules must still be written together in one place. What is often desired in practice, however, is something stronger: the ability to *compile* mutually recursive modules separately.

Separate compilation is supported in ML via the functor mechanism. If a module `A` depends on a module `B` of signature `SIG_B`, then `A` can be separately compiled from `B` by defining it as a functor `F_A` parameterized over a module of signature `SIG_B`. Later, when the program components are to be linked together, the `A` module can be defined by applying `F_A` to the actual `B` module.

Suppose that we try compiling `Expr` and `Bind` separately, as shown in Figure 5.11, by turning each module into a functor that is parameterized over the recursive variable `X`. To link the modules, we write a recursive module whose substructures `Expr` and `Bind` are defined by instantiating the respective functors with the actual recursive variable `X`.

One problem with this approach is that, since ML is call-by-value, the occurrences of the recursive variable `X` as an argument to `F_Expr` and `F_Bind` will result in an attempt to evaluate `X` before the body of `Link` is finished evaluating. Thus, according to the backpatching semantics for recursive modules, the evaluation of the `Link` module will raise an error indicating that the recursion is unsafe. The issue here is that, under the dynamic semantics I described in Section 5.2.1, the memory location to which `X` is bound is implicitly dereferenced wherever `X` appears in the recursive module body. In the separate compilation scenario, though, this is clearly not the intended semantics. Instead, where `X` is passed to `F_Expr` and `F_Bind`, we would like to pass the memory location to which `X` is bound *without* dereferencing it.

There is a relatively simple way of addressing this problem, which is to make the act of dereferencing a recursive variable explicit. First, we introduce a new signature $\mathsf{maybe}(\mathrm{S})$ describing modules of signature S that may or may not be defined. A value of signature $\mathsf{maybe}(\mathrm{S})$ is a memory location whose contents (of signature S) are in the process of being computed. Then, rather than treating the recursive variable X in $\mathsf{rec}(\mathrm{X}:\mathrm{S}.\,\mathrm{M})$ as a (non-value) module expression of signature S, we can treat X as a value of signature $\mathsf{maybe}(\mathrm{S})$ that must be explicitly dereferenced by writing $\mathsf{fetch}(\mathrm{X})$.

Given this new form of signature, we can rewrite the recursive module typing rule from Section 5.2.3 as follows:

$$\frac{\Gamma, \mathrm{X}{:}\mathsf{maybe}(\mathrm{S}) \vdash \mathrm{M} :_{\kappa} \mathrm{S}}{\Gamma \vdash \mathsf{rec}(\mathrm{X} : \mathrm{S}.\, \mathrm{M}) :_{\kappa} \mathrm{S}}$$

With this semantic modification, the `Link` module in Figure 5.11 will no longer attempt to dereference `X`. In order for the functor applications defining `Expr` and `Bind` to be well-formed, though, we must: (1) change the argument signature of `F_Expr` and `F_Bind` to maybe(`EXPR_BIND`), and (2) replace references to `X` in the bodies of those functors with fetch(`X`).

Unfortunately, this new typing rule still runs afoul of the double vision problem. In particular, just as when `Expr` and `Bind` were defined together in Figure 5.10, there is no way in the body of `F_Expr` to connect `X.Expr.t` with the implementation of `t` that the body of `F_Expr` provides. Separate compilation certainly does not make the problem any easier.

5.3 Existing Approaches to Recursive Modules

In Sections 5.3.1–5.3.3, I describe the existing proposals for extending ML with recursive modules. I will discuss the strengths and weaknesses of these proposals in terms of how they cope with the issues surveyed in the previous section, especially the double vision problem. In Sections 5.3.4 and 5.3.5, I survey some alternative approaches to cross-module recursion that involve replacing ML's mechanisms for modular composition with something significantly different.

5.3.1 A Foundational Account

Crary, Harper and Puri [6] (hereafter, CHP) give a foundational, type-theoretic account of the recursive module problem. They are responsible for a number of the important ideas presented in Section 5.2. In particular, they introduced the concept of recursively dependent signatures, and were the first to identify the double vision problem (although they do not call it that).

The actual type system that CHP define is more of a language sketch than it is a full-fledged ML extension. It is built on top of Harper, Mitchell and Moggi's phase-distinction calculus [30]. It supports translucency, as I did in Chapter 3, by extending the HMM core language with singleton kinds, but it does not support any form of sealing. The type system also includes equi-recursive type constructors, but the question of decidability of type checking is left to future work.

The CHP type system employs a fixed-point semantics for recursion, which is made possible by their restriction that the body of a recursive module be "valuable," *i.e.,* effect-free and terminating. This places a significant limitation on the kinds of recursive modules one can write. For example, the recursive module in Figure 5.6 is not expressible.

CHP's approach to the double vision problem is to require that the declared signature of a recursive module be transparent. In order to allow `datatype` specifications to occur in a transparent signature, CHP rely on a "transparent interpretation" of ML `datatype`'s. Under such an interpretation, a `datatype` specification for `t` specifies the type `t` as being transparently equal to a particular recursive type, instead of leaving the identity of `t` abstract. First of all, it is difficult to reconcile the transparent interpretation of `datatype`'s with ML's semantics, in which `datatype`'s are abstract types. (See Vanderwaart *et al.* [79] for a thorough analysis of this issue.) Moreover, requiring the signature of a recursive module to be transparent stymies the enforcement of any data abstraction between mutually recursive submodules, like `Expr` and `Bind`.

CHP propose the use of functors for separate compilation of recursive modules, in the manner of the example from Figure 5.11. As I argued in Section 5.2.4, this approach only works if we introduce a distinction between the types of recursive variables and the types of ordinary variables, and make the dereferencing of recursive variables explicit. CHP do not do this, and consequently their example of how to support separate compilation does not typecheck. It would not be hard to fix this problem—by extending their system with a new $\mathsf{maybe}(\mathrm{S})$ signature as I proposed in Section 5.2.4. Still, due to the double vision problem, their approach would only enable the separate compilation of recursive modules with transparent signatures.

In short, CHP's main contribution has been to provide a conceptual framework in which the recursive module problem may be discussed coherently. Their main limitation is that they do not resolve the double vision problem in a satisfactory way.

5.3.2 Moscow ML

Russo [66] has implemented a recursive module extension to ML in the context of the Moscow ML compiler [56]. This is the first recursive module extension to be actually implemented. Inspired by CHP, Russo's extension introduces a recursive module construct and an rds construct. The rds construct is subject to the relaxed form of the dynamic-on-static restriction (described in Section 5.2.2), in which recursive references in transparent type specifications are permitted, but only if they are acyclic. Russo's rds construct is written $\texttt{rec}(\mathrm{X}\texttt{:}\mathrm{S}_1)\mathrm{S}_2$, where S_1 serves as the signature of X when checking the well-formedness of S_2. It is not clear why the programmer is required to write S_1 instead of simply having the type system infer it from S_2.[4]

Russo's recursive module construct has the same form as the one studied in Section 5.2. He employs a backpatching semantics and relies on dynamic checks to ensure that the recursion is safe. The body of a recursive module may therefore have arbitrary computational effects, and these effects are guaranteed to only happen once.

Russo's typing rule for $\mathsf{rec}(\mathrm{X}:\mathrm{S}.\,\mathrm{M})$ differs from any of the typing rules considered in Section 5.2. First, it allows M to provide components that are not specified explicitly in S. In other words, S is used merely as a "forward declaration" of those components that will be needed recursively; it does not represent the signature of the whole module. Second, M is required to be coercible[5] to the signature $\mathfrak{S}_\mathrm{S}(\mathrm{X})$. This requirement is Russo's way of avoiding the double vision problem: if M is coercible to $\mathfrak{S}_\mathrm{S}(\mathrm{X})$, then for any type component `t` specified by S, the definition for `t` given by M must coincide with the type `X.t`.

Russo's solution to double vision is, for the most part, the same as requiring that S be transparent. To illustrate this, suppose that we try to write the `ExprBind` example in Moscow ML. Figure 5.12 shows a first attempt, in which the declared signature `EXPR_BIND` hides the `datatype` definitions of `Expr.t` and `Bind.t`. Since there is no way to connect the `datatype` definitions in the body to the abstract types `X.Expr.t` and `X.Bind.t`, the module does not typecheck. Generally speaking, under Russo's typing rule, if the declared signature contains an opaque specification of the form `type u`, the only way the body of the recursive module will be allowed to define `u` is by writing `type u = X.u`, in which case `u` never gets defined! Thus, opaque type specifications in declared signatures are essentially useless.

In order to write recursive modules like `Expr` and `Bind` in Moscow ML, the programmer is forced to expose the `datatype` definitions of `Expr.t` and `Bind.t` in the declared signature, as shown in Figure 5.13. We rely in this version of `ExprBind` on a little-known feature of Standard

[4] As, for example, Leroy's extension to O'Caml does (described below).

[5] According to ML's notion of coercive signature matching, formalized in Section 9.3.4.

```
signature EXPR_BIND =
sig
  structure Expr : sig type t ... end
  structure Bind : sig type t ... end
end
structure ExprBind = rec (X : EXPR_BIND)
struct
  structure Expr = struct datatype t = ... end
  (* Type error: Expr.t ≢ X.Expr.t *)
  structure Bind = struct datatype t = ... end
  (* Type error: Bind.t ≢ X.Bind.t *)
end
```

Figure 5.12: `Expr` and `Bind` in Moscow ML: First Try

```
signature EXPR_BIND =
rec (X : sig structure Expr : sig type t end
             structure Bind : sig type t end end)
sig
  structure Expr : sig datatype t = ... end
  structure Bind : sig datatype t = ... end
end
structure ExprBind = rec (X : EXPR_BIND)
struct
  structure Expr = struct datatype t = datatype X.Expr.t ... end
  structure Bind = struct datatype t = datatype X.Bind.t ... end
end
```

Figure 5.13: `Expr` and `Bind` in Moscow ML: Second Try

ML, namely the `datatype` replication binding. By writing `datatype t = datatype X.Expr.t`, the `Expr` module defines the type `t` as equivalent to `X.Expr.t`, and also binds local copies of all of `X.Expr.t`'s data constructors. With this binding, the double vision problem is clearly avoided, and the example typechecks.

The astute reader may have noticed that, even in this successful attempt at `Expr` and `Bind`, the types `Expr.t` and `Bind.t` are still never defined anywhere. The declared signature provides `datatype` specifications for them, which indicates what the types of their data constructors are, but the types themselves are abstract because `datatype` specifications are opaque. Russo is not clear on this point. My interpretation is that, when a `datatype` specification (or any opaque type specification) appears in the declared signature of a recursive module, the compiler implicitly defines the type in some canonical way that matches the specification. Since the type is abstract, the program will typecheck regardless of which canonical implementation is chosen. See Section 9.3.3 for further discussion of how to construct canonical implementations of certain specifications.

To summarize, Russo's extension makes it possible to encode all of the motivating examples from Section 5.1, but prevents the programmer from hiding type information between mutually recursive modules. This limitation is alleviated slightly by the fact that the body of a recursive module is

```
functor F_Expr (X : EXPR_BIND) = ...
...
functor F_Bind (X : EXPR_BIND) = ...
...
structure ExprBind = rec (X : EXPR_BIND)
struct
  structure EtaX = struct
    structure Expr = struct
      datatype t = datatype X.Expr.t
      fun eval e = X.Expr.eval e
      ... (* define eta-expansion for every function *)
    end
    structure Bind = ... (* similarly for Bind *)
  end
  structure Expr = F_Expr(EtaX)
  structure Bind = F_Bind(EtaX)
end
```

Figure 5.14: Separate Compilation of `Expr` and `Bind` in Moscow ML

allowed to define additional type components that are not specified in the declared signature. For example, `Expr` and `Bind` are free to define type components other than `t` and to hold their identities abstract, but only if those type components do not need to be forward-declared in the signature of `X`. It is hard to say how useful this flexibility would be in practice.

Lastly, with regard to separate compilation, Russo proposes the use of functors. Unlike CHP, he correctly observes that the separate compilation scenario of Figure 5.11 does not work. Instead, he suggests an alternative solution that only works when all the value components of the recursive variable `X` are functions. His solution, shown in Figure 5.14, is to define (in the linking module) a structure called `EtaX` whose type components are copies of the type components of `X` and whose value components are eta-expansions of the value components of `X`. The modules `Expr` and `Bind` can then be defined by applying `F_Expr` and `F_Bind`, respectively, to `EtaX` instead of `X`. While this solution does work in the particular case of `Expr` and `Bind`, it does not constitute a general solution. Moreover, from an aesthetic standpoint, it is not much of an improvement on the recursive module workarounds discussed in Section 5.1.

5.3.3 O'Caml

Leroy has implemented recursive modules as an experimental extension to the O'Caml language, available in version 3.07 of the compiler and later [41]. O'Caml does not have a formal definition, and neither does its recursive module extension, but Leroy has made an informal description of his extension available on the web [44].

Leroy introduces rds's via a recursively dependent `module rec` specification. Figure 5.15 shows how the `EXPR_BIND` signature would be written in O'Caml. Similarly to Moscow ML, Leroy avoids the need for equi-recursive type constructors by prohibiting cyclic dependencies from occurring in transparent type specifications.

For recursive modules, Leroy introduces a `module rec` binding, whose syntax is almost identical

```
module type EXPR_BIND =
sig
  module rec Expr : sig
    type t
    val makeLetExpr : Bind.t * t -> t
    ...
  end
  and Bind : sig
    type t
    val makeValBind : var * Expr.t -> t
    ...
  end
end
```

Figure 5.15: Signature for `Expr` and `Bind` in O'Caml

to the `structure rec` binding I used in the motivating examples of Section 5.1. The binding

$$\texttt{module rec } X_1 \texttt{ : } S_1 \texttt{ = } M_1 \texttt{ and ... and } X_n \texttt{ : } S_n \texttt{ = } M_n$$

defines a bundle of n mutually recursive modules, $M_1,\ldots,M_n$, that refer to one another as $X_1,\ldots,X_n$. The binding is evaluated according to backpatching semantics. Leroy imposes a restriction on recursive modules, though, that enables him to implement backpatching more efficiently. Specifically, in Leroy's semantics, a module may only be depended on recursively if all of the value components in its interface have "pointed" type, *i.e.,* they are all functions or lazy computations, for which there is a canonical bottom ($\perp$) element.[6] As a result, the recursive modules `A` and `B` in Figure 5.6 are not encodable in O'Caml because both modules have value components of the non-pointed type `bool ref`. On the other hand, O'Caml does allow one to write all of the motivating examples from Section 5.1.

To see why Leroy's restriction enables more efficient implementation of backpatching, consider the single binding `module rec` X `:` S `=` M. Under the general backpatching semantics, the contents of the location X are uninitialized during the evaluation of M. Thus, in order to ensure type safety, we must prevent X from being dereferenced until its contents have been initialized via backpatching. There are various ways to ensure that X does not get dereferenced, but they all involve some run-time cost.[7] However, if S only specifies values of pointed type, then there is a canonical way of constructing a "dummy" module N of signature S such that, if any of the functions in N are applied (or any of the lazy computations in N are forced), an "unsafe recursion" exception will be raised. This dummy module N gives us a way of safely initializing the contents of X before evaluating M, so there is no need to dynamically protect against the dereferencing of X.

Aside from the pointed-type restriction, Leroy's typing rule for recursive modules is precisely the simple typing rule suggested at the beginning of Section 5.2.3. As we know, this typing rule runs afoul of the double vision problem. Leroy's solution is somewhat *ad hoc* and is easiest to explain in terms of the `ExprBind` example. When we are typechecking the body of `Expr` and encounter

[6] To be precise, Leroy requires that there be *some* evaluation ordering of the recursive modules, $M_1,\ldots,M_n$, such that all "forward" references (*i.e.,* references to modules that have not been evaluated yet) are to modules whose components have pointed type. For more in-depth discussion, see also Hirschowitz, Leroy and Wells [36].

[7] See the beginning of Chapter 7 for more discussion of this point.

```
module type S = sig
  type t
  val f : t -> t
end
module rec X : sig module A : S end = struct
  module A : S = struct
    type t = C of int
    let f (x : t) : t = X.A.f x
  end
end
```

Figure 5.16: Strange Behavior of O'Caml Recursive Module Typechecking

the `datatype` definition of `t`, the O'Caml typechecker adds to the context the assumption that `t` is equivalent to `X.Expr.t`. This enables the body of `Expr` to know that `X.Expr.t` is implemented as `t`, even though the signature of `X` does not reveal this. It also allows `Expr` and `Bind` to be written in O'Caml without forcing their `datatype` definitions to be exposed in their signatures.

Leroy's approach to the double vision problem is admirable in that, of the existing approaches, it comes closest to realizing the Alice-and-Bob intuition that I gave in Section 5.2.3 for describing how recursive module typechecking should interact with data abstraction. Yet there are two serious problems with it. One is that it only appears to work for type components that are implemented internally by `datatype` definitions.[8] For instance, if `Expr.t` were defined in the body of `Expr` by an ordinary type definition like `type t = int`, then the typechecker would *not* add to the context the assumption that `X.Expr.t` was equivalent to `int`. Thus, O'Caml's recursive modules do not avoid the double vision problem in general. This runs counter to intuition and gives preferential treatment to types implemented by `datatype` definitions for no clear reason.

Another problem is that no type-theoretic explanation is provided for what it means to "add a type-equivalence assumption to the context" during the typechecking of the recursive module body. In Section 5.4.2, I will provide such an explanation in the context of my own recursive module proposal. In the absence of such an explanation, though, the O'Caml approach can be easily shown to exhibit strange behavior. For instance, consider the simple, contrived recursive module shown in Figure 5.16. The O'Caml type definition `type t = C of int` corresponds to `datatype t = C of int` in SML. According to the semantics Leroy describes, once this type definition is processed, we are able to observe that `t` is equivalent to `X.A.t`. This is important because the well-formedness of `f` depends on the equivalence of `t` and `X.A.t`. However, if the `t` annotations on the domain and range types of the function `f` are replaced by `X.A.t` (or omitted altogether), the O'Caml typechecker complains that the type of `f` is `X.A.t -> X.A.t` and that this type is not equivalent to `t -> t`. Given Leroy's informal semantics, this type error does not make any sense.

In conclusion, Leroy's O'Caml extension improves on previous work in that it allows for recursive modules to hide type information from one another. However, in order to avoid the double vision problem, the types whose identities are hidden must be defined internally as `datatype`'s. Leroy's extension also supports more efficient implementation of recursive modules, but this comes at the cost of not being able to write perfectly reasonable recursive modules such as the one in Figure 5.6. Leroy does not address the issue of separate compilation.

[8]Note: `datatype` definitions are written in O'Caml using the same keyword (`type`) as ordinary type definitions, but for expository purposes, I prefer to stick with SML syntax and refer to them as `datatype` definitions.

5.3.4 Units

As is surely evident at this point, extending ML with recursive modules is a difficult endeavor. A number of researchers have thus investigated ways of replacing ML's notion of *module* with some alternative mechanism for which recursive linking is the norm and hierarchical linking a special case. The two best-known alternative mechanisms are *units* and *mixins*.

Flatt and Felleisen [20] introduced units as the foundation of a module system for the Scheme language. At first glance, a unit appears to be roughly like an ML functor: it requires some imports and provides some exports. Unlike functors, however, two units can be linked recursively to form a compound unit, with the exports of each unit being used to satisfy the import requirements of the other unit. Eventually, once enough units have been compounded together that the resulting unit requires no more imports, the compound unit can be "invoked," resulting in the execution of its initialization routine.

Aside from the ability to link units recursively, one fundamental difference between units and ML modules is that the interdependencies between units can only be described externally to the modules themselves. In ML, the implementor of a module `A` may wish to use the implementation of sets provided by a particular module `FastSet`, and the implementor can indicate this by referring to `FastSet` directly and projecting out its components. With units, the implementor of `A` can only specify that it wants *some* set module as an import. The burden of ensuring that `A` actually gets linked to `FastSet` is shifted to whoever wants to use `A`. It is worth noting that the unit approach of external linking is encodable in ML. In particular, one can write every module in an ML program as a functor parameterized over its imports, and use functor application to perform explicit linking. This is referred to as programming in "fully functorized" style. ML's direct implementation-on-implementation dependencies, on the other hand, are not encodable using units.

Another fundamental difference is that units are first-class. They may be compiled separately and then linked dynamically based on run-time information. In the context of a dynamically-typed language like Scheme, a first-class module system is ideal, and units have been deployed successfully in the development of the DrScheme programming environment [19]. In the context of ML, though, where modules have type components, having a purely first-class module system makes it difficult to track the propagation of type information effectively.[9] Flatt and Felleisen consider the extension of Scheme's units to include ML-style type definitions, but their extension does not include support for translucency in unit interfaces. Translucency is a key feature of the ML module system that is not worth abandoning just in order to support first-class, recursive modules.

Having said that, it is also worth pointing out that Flatt and Felleisen's extension of units to include type components does not seem to suffer from double vision. In particular, one way to phrase the double vision problem is that the types classifying a recursive module's value imports may need to depend on the abstract types that the module exports. With recursive ML-style modules, it is not obvious how to allow a module's imports to depend on types that the module has not yet defined. With "type-enhanced" units, though, this is not an issue. A unit must give explicit names to both its type imports and its type exports, and the types that classify a unit's value imports and exports are allowed to refer to any of the type imports or exports. As a result, double vision is avoided. This comes, however, at the cost of a rather restrictive and unwieldy syntax, and of a type system whose inference rules are very large and complex. It remains an interesting direction for future work to determine whether there exists a simple type-theoretic way of explaining what units are doing, as well as a compromise design that balances the elegance and translucency of ML modules with the unit approach to avoiding double vision.

[9] See Section 1.2.3 for further discussion of this point.

5.3.5 Mixins

The term "mixin" refers to a variety of modularity constructs, all of which are intended to support recursive composition in something resembling an object-oriented (OO) style. Different notions of mixins attempt to adapt different aspects of OO languages to a functional-language setting.

It is well-known that functional and OO languages provide orthogonal forms of extensibility. Functional languages make it easy to define new functions that operate over a `datatype`, while OO languages make it easy to extend an existing `datatype` with new branches. Duggan and Sourelis [15] propose a `mixin` module extension to SML, whose goal is to provide OO-style extensibility by making it possible to split the definition of a *single* function or `datatype` across module boundaries. A `mixin` module is like an ML structure but is divided into three parts: a prelude, a body, and an initialization section. The body may only contain only `datatype` and `fun` bindings. When two `mixin` modules are composed, `datatype` (or `fun`) bindings in the bodies of each `mixin` that define types (or functions) of the same name are merged together into a single `datatype` (or `fun`) binding. The other sections of the `mixin` are not restricted, and may have arbitrary effects, but they are not involved in the recursive merging.

The functionality provided by Duggan and Sourelis' `mixin` modules is orthogonal to the functionality of recursive modules studied in this chapter. I have been concerned here with the ability to split mutually recursive functions and `datatype`'s into separate modules, not to split different cases of a single function or `datatype` definition into separate modules. In later work [16], the authors propose the introduction of a second "`mixlink`" mechanism for `mixin` composition that is closer in aim to the form of recursive module linking studied here. This `mixlink` construct is accompanied, though, by a rather baroque set of restrictions on the structure of the participating `mixin` modules. An advantage of the recursive module extension I describe in Section 5.4 is that the restrictions it imposes on the programmer are easy to explain. It is difficult to comment further on Duggan and Sourelis' `mixlink`, as the authors describe its semantics only informally and provide no implementation with which to experiment.

In more recent work, Duggan [14] has developed a language of "recursive DLL's" that diverges significantly from the ML module system. This is an intermediate language, not a source-level language, whose main goal is to support dynamic linking and loading of shared libraries. Duggan addresses the double vision problem in a manner similar to O'Caml, but the problem is simplified by the fact that Duggan's language only supports opaque `datatype`-like bindings, not transparent `type` bindings.

Ancona and Zucca [4] propose a very different kind of mixin module construct in their CMS calculus. CMS's mixins are actually very similar to Flatt and Felleisen's units. The main difference is that CMS also supports "overriding with late binding" of module components, a feature commonly associated with OO-style inheritance. CMS itself is somewhat impractical: it is purely functional and call-by-name, and its mixin modules do not contain type components or provide the ability to define abstract data types. These simplifications allow CMS to avoid dealing with the complex issues surrounding the interaction of recursion with effects and the double vision problem. There have been several proposals for making CMS more realistic—adding support for computational effects [3], and transferring CMS to a call-by-value setting [35]—but none has yet attempted to extend mixin modules with type components.

5.4 A New Approach

In this section, I describe at a high level my own proposal for extending ML with recursive modules. The main distinction between my proposal and the existing proposals described in Section 5.3 is that mine offers a more satisfying solution to the double vision problem.

5.4.1 Overview

Like the existing recursive module extensions, my extension introduces two new constructs: one for recursive modules and one for recursively dependent signatures. For coherence of notation, I will write these constructs using syntax similar to Moscow ML's. Recursive modules have the form `rec`(X:S)M, and recursively dependent signatures have the form `rec`(X)S.[10]

The dynamic semantics for the recursive module construct `rec`(X:S)M is a straightforward backpatching semantics. I do not make any restrictions on the declared signature as O'Caml does, nor do I attempt to statically detect that the recursion is safe, *i.e.,* that X will not be dereferenced during the evaluation of M. The problem of statically ensuring safe recursion will be considered in Chapter 7.

For recursively dependent signatures `rec`(X)S, I impose the dynamic-on-static restriction described in Section 5.2.2. That is, I do not allow any recursive references to X within the specifications of type components in S, even if such references do not introduce any cyclic type specifications. The main reason for this is simplicity. Dynamic-on-static rds's constitute a relatively straightforward extension to my type system for modules. (For more details, see Chapter 6.) Finding a way to type-theoretically account for O'Caml's more general rds's—in which recursive dependencies in type specifications are permitted so long as they essentially acyclic—remains a worthwhile direction for future work. Nevertheless, the more restricted form of rds that I provide is sufficient to encode all of the motivating examples given in this chapter.

Where my proposal differs most from the existing designs is in the typechecking of recursive modules. Given a recursive module `rec`(X:S)M, I make two simple restrictions on the body M, described below. As long as M obeys these restrictions, my static semantics ensures that the double vision problem *never* arises.

The first restriction I place on M is a kind of dynamic-on-static restriction analogous to the one imposed on rds's. For an rds `rec`(X)S, references to X may occur in `datatype` specifications, but not in transparent type specifications. Similarly, in the recursive module `rec`(X:S)M, I allow X to be referenced inside `datatype` bindings, but not inside transparent `type` bindings, like `type t =` C.

This restriction is motivated by the desire to avoid the double vision problem in all cases. Suppose that the following code, in which a recursive reference to `X` occurs in a transparent `type` binding, were permitted:

```
rec (X : sig type t ... end)
struct
  type t = int * X.t
  ...
end
```

In order for my semantics to avoid double vision here, it would have to somehow ensure that the body of the module is typechecked in a context where `X.t` = `t` = `int * X.t`. This cyclic type equivalence would require the introduction of equi-recursive types into the core language. The dynamic-on-static restriction allows me to get by without extending the core language.

[10]Note that, unlike Moscow ML, I do not require any signature annotation on X in the rds construct.

The second restriction on the recursive module body is that it be phase-separable, *i.e.,* that it have purity classifier `P` according to the type system of Chapter 4. Thus, the only way that data abstraction may be enforced in the body of the recursive module is through the use of *basic* sealing. The reason for this restriction is that, in my formalization of recursive module typechecking, I need to be able to phase-split the recursive module body and extract its static part. In general, this is only possible if the body is separable.

How significant are these restrictions? It is difficult to say without more experimentation. At the very least, they do not pose a problem for any of the motivating examples given in Section 5.1. With only superficial syntactic modifications, all of these examples are expressible in my extension.

Moreover, in return for its restrictions, my semantics improves on both the Moscow ML and O'Caml approaches in terms of its support for data abstraction within recursive modules. For instance, consider our running `ExprBind` example. In Moscow ML, `Expr.t` and `Bind.t` must have their `datatype` definitions exposed in the declared signature in order for the example to even typecheck. In O'Caml, `Expr.t` and `Bind.t` need not have their `datatype` definitions exposed, but in order to avoid the double vision problem, they *must* be implemented by `datatype` bindings. Under my approach, `Expr.t` and `Bind.t` need not have their definitions exposed, *and* they may be implemented either by `datatype` bindings or by transparent `type` bindings. Furthermore, my proposal is formally defined, whereas the O'Caml extension is not.

Finally, like the existing recursive module extensions, my proposal does not, in its current form, provide support for separate compilation of mutually recursive modules. The reason is that, in order to avoid the double vision problem, the body of a recursive module is typechecked in a very special way, as I describe below. If two mutually recursive modules `A` and `B` are compiled separately, then whatever mechanism we use to compile them must employ this special form of typechecking as well. (In particular, as discussed in Section 5.2.4, if we just try to use functors, we run into the double vision problem all over again.) Thus, in order to support separate compilation of recursive modules, we must introduce a new separate compilation mechanism that is aware of recursive module typechecking. In Chapter 10, I propose as future work to incorporate such a mechanism into the design of a general compilation management language, built on top of ML.

5.4.2 Elaboration of Recursive Modules

Recall that, under the Harper-Stone approach to formalizing ML, there are two languages: the external language (EL), namely ML (or the extension of it that we are defining), and the internal language type system (IL), into which the EL is elaborated. The purpose of the IL is to encapsulate in type theory as much of ML semantics as is possible.

After studying recursive modules for some time, the only way I have found to cleanly avoid the double vision problem in type theory is the original approach I suggested in Section 5.2.3, namely to require that the declared signature of a recursive module be transparent. Thus, in the IL of my new ML dialect, I provide a recursive module construct $\mathsf{rec}(\mathrm{X}:\mathbb{S}.\,\mathrm{M})$, which syntactically restricts the declared signature to be transparent. The typing rule for $\mathsf{rec}(\mathrm{X}:\mathbb{S}.\,\mathrm{M})$ is as follows:

$$\frac{\Gamma, \mathrm{X}{:}\mathsf{maybe}(\mathbb{S}) \vdash \mathrm{M} :_{\mathtt{P}} \mathbb{S}}{\Gamma \vdash \mathsf{rec}(\mathrm{X}:\mathbb{S}.\,\mathrm{M}) :_{\mathtt{P}} \mathbb{S}}$$

This is the same as the typing rule given in Section 5.2.4, except that here I also require that M be pure/separable.[11] This purity condition poses no burden, for if M is impure and has the transparent signature $\mathbb{S}$, then we can always coerce M's purity classifier to `P` by writing $\mathsf{purify}(\mathrm{M})$.

[11] I have taken the approach of binding X with the signature $\mathsf{maybe}(\mathbb{S})$, instead of employing a special recursive

```
signature CSS = ρX. sig
  structure Expr : sig
    datatype t = ... | LetExpr of X.Bind.t * t | ...
  end
  structure Bind : sig
    datatype t = ... | ValBind of var * Expr.t | ...
  end
end
```

Figure 5.17: Closed Static Signature of `Expr` and `Bind`

In contrast, my EL recursive module construct `rec(`X`:`S`)`M does not require S to be transparent. This shifts the burden of avoiding the double vision problem onto the elaboration algorithm. The goal of the final section of this chapter is to provide an informal understanding of how recursive module elaboration works. The formal details are presented in Chapter 9.

The "Easy" Case

Suppose that we are given a recursive EL module `rec(`X`:`S`)`M, whose body M obeys the dynamic-on-static and separability restrictions described above. Let us begin by considering how this module is elaborated in the "easy" case where M does not contain any explicit uses of sealing. That is, assume that the only abstract types in M arise from `datatype` definitions. An example of such a module is the version of `ExprBind` shown in Figure 5.12. The figures accompanying the following discussion illustrate the output of elaboration for this version of the `ExprBind` module.

In the absence of sealing, the basic idea in elaborating `rec(`X`:`S`)`M is to construct a transparent signature $\mathbb{S}'$ that fills in the opaque type specifications of S with the definitions of those type components provided by M. In essence, $\mathbb{S}'$ will be equivalent to $\mathfrak{s}_{\mathrm{S}}(\mathsf{Fst}(\mathrm{M}))$. If we use $\mathbb{S}'$ as the declared signature of X (instead of S), then we will avoid double vision because we will be able to observe that $\mathsf{Fst}(\mathrm{X})$ is equivalent to $\mathsf{Fst}(\mathrm{M})$.

Unfortunately, in reality, we cannot directly use $\mathfrak{s}_{\mathrm{S}}(\mathsf{Fst}(\mathrm{M}))$ as the declared signature of X, because M may contain `datatype` bindings with recursive references to X. The solution is to first define a module, called `Static`, which resolves the recursive dependencies among M's type components while ignoring its term components. We can then use `Static` in place of $\mathsf{Fst}(\mathrm{M})$ in the declared signature of X.

To compute `Static`, we begin by generating a signature R, which contains as precise a specification of M's type components as possible and ignores its value components. That is, for every `type` *binding* in M, `type t =` C, there will be a `type` *specification* in R of the form `type t =` C; for every `datatype` binding in M, the corresponding `datatype` specification will appear in R as well. For every `val` or `fun` binding in M, no specification appears in R. I will refer to R as the "static signature" of M.

The static signature R may have free recursive references to X. However, thanks to the dynamic-on-static restriction on M, these recursive references may only appear within `datatype` specifications, not within transparent `type` specifications. Consequently, the recursively dependent signature

variable binding $\mathrm{X}\uparrow\mathbb{S}$, both to simplify the module meta-theory and make the construct more amenable to the eventual support of separate compilation. Correspondingly, the dereferencing of X must be indicated explicitly in the body M by writing $\mathsf{fetch}(\mathrm{X})$.

```
structure Static :> CSS =
let
    datatype Expr_t = ... | LetExpr of Bind_t * t | ...
         and Bind_t = ... | ValBind of var * Expr_t | ...
in
    struct
      structure Expr = struct
        datatype t = datatype Expr_t
      end
      structure Bind = struct
        datatype t = datatype Bind_t
      end
    end
end
```

Figure 5.18: Canonical Implementation of `Expr` and `Bind`'s Closed Static Signature

```
structure ExprBind = rec (X : 𝔖EXPR_BIND(Static))
struct
  structure Expr = struct
    datatype t = ... | LetExpr of X.Bind.t * t | ...
    (* Double vision: t ≢ X.Expr.t *)
  end
  structure Bind = struct
    datatype t = ... | ValBind of var * Expr.t | ...
    (* Double vision: t ≢ X.Bind.t *)
  end
end
```

Figure 5.19: Elaboration of `Expr` and `Bind`: First Try

ρX.R is guaranteed to be well-formed. I will refer to this rds as the "*closed* static signature" of M. The closed static signature of `ExprBind` is shown in Figure 5.17.

Given the closed static signature ρX.R of M, we may now define `Static` as the "canonical" module implementing this signature. The details of computing canonical implementations of signatures are given in Section 9.3.3, but the basic idea is straightforward. We first resolve the recursive references to X in the `datatype` specifications of R by joining all the type components of the module in a big `datatype` binding. Once we have done this, we can copy the types into the appropriate substructures. Figure 5.18 illustrates this construction in the case of `Expr` and `Bind`.

Having defined `Static`, we can now use $\mathfrak{S}_{\mathrm{S}}$(`Static`) as the declared signature of the recursive module. There is one last tricky point, however, regarding `datatype`'s. Figure 5.19 shows what goes wrong in the `ExprBind` example if we use $\mathfrak{S}_{\mathrm{S}}$(`Static`) as the declared signature but leave the recursive module body unchanged. Specifically, any `datatype` bindings in the body will generate fresh abstract types that are not equivalent to the corresponding components of `X` or `Static`. The solution is simple: replace every `datatype` binding in the recursive module body with a `datatype` replication binding that copies the corresponding `datatype` component from `Static` instead of

```
structure ExprBind = rec (X : 𝔖EXPR_BIND(Static))
struct
  structure Expr = struct
    datatype t = [datatype Static.Expr.t]
  end
  structure Bind = struct
    datatype t = [datatype Static.Bind.t]
  end
end
```

Figure 5.20: Elaboration of `Expr` and `Bind`: Second Try

```
structure ExprBind = rec (X : 𝔖EXPR_BIND(Static))
struct
  structure Expr :> sig type t ... end =
    struct ... (* same as in Figure 5.20 *) ... end
  structure Bind :> sig type t ... end =
    struct ... (* same as in Figure 5.20 *) ... end
end
```

Figure 5.21: Elaboration of `Expr` and `Bind` With Sealing: First Try

defining a fresh one. Figure 5.20 shows how this is done for `Expr` and `Bind`.

This completes the discussion of recursive module elaboration in the case where M contains no sealing. In the interest of expository simplicity, I have glossed over a few technical issues. For instance, it is possible that the recursive module body M defines more type components than are specified in the declared signature S or that it defines them in a different order than S does. In this case, while M will be coercible to S according to ML's notion of signature matching, the signature $\mathfrak{S}_{\mathrm{S}}$(`Static`) will not be well-formed because the kind of Fst(`Static`) will not be an IL subkind of Fst(S). This is not a problem in practice, however. As long as M is coercible to S, we can generate a type constructor `Static'` that copies and reorders (some of) the type components of `Static` in order to match Fst(S) precisely. We can then use $\mathfrak{S}_{\mathrm{S}}$(`Static'`) instead of $\mathfrak{S}_{\mathrm{S}}$(`Static`) for the declared signature of the recursive module. I will leave discussion of other technical subtleties until Section 9.3.8, in which recursive module elaboration is formalized.

The General Case

We now turn attention to the general problem of elaborating `rec`(X:S)M, where the body M may contain uses of (basic) sealing. To make the problem concrete, consider how we might elaborate our key motivating example, namely the version of the `ExprBind` module in which `Expr` and `Bind` are sealed with signatures that hide the data constructors of `Expr.t` and `Bind.t`.

A first attempt at elaborating this version of `ExprBind` is shown in Figure 5.21. The static part of `ExprBind` is the same as it was before, so the definitions of `Static` and `CSS` remain unchanged from their definitions in Figures 5.17 and 5.18. The only change here is that the definitions of `Expr` and `Bind` are sealed with opaque signatures. This is problematic because it means that the body of the recursive module does not match the transparent declared signature $\mathfrak{S}_{\texttt{EXPR\_BIND}}$(`Static`).

```
structure ExprBind = rec (X : 𝔖EXPR_BIND(Static))
struct
  structure Expr :> sig type t = Static.Expr.t ... end =
    struct ... (* same as in Figure 5.20 *) ... end
  structure Bind :> sig type t = Static.Bind.t ... end =
    struct ... (* same as in Figure 5.20 *) ... end
end
```

Figure 5.22: Elaboration of `Expr` and `Bind` With Sealing: Second Try

```
signature CSS = sig
  structure Expr : sig type t = C end
  structure Bind : sig type t = D end
end
structure Static :> CSS = ...

structure ExprBind = rec (X : 𝔖EXPR_BIND(Static))
struct
  structure Expr :> sig type t = Static.Expr.t ... end =
    struct type t = C ... end
  structure Bind :> sig type t = Static.Bind.t ... end =
    struct type t = D ... end
end
```

Figure 5.23: Problematic Elaboration of Modified `ExprBind` Example

Figure 5.22 shows how the elaborator can address this issue by modifying the sealing signatures so that they expose the equivalence of their t components with `Static.Expr.t` and `Static.Bind.t`, respectively. This technique is similar to the one used to transform `datatype` definitions in the recursive module body (cf. Figure 5.20). Here, however, it results in a loss of data abstraction. In particular, the implementation of `Bind` in Figure 5.22 is able to observe that `Expr.t` is equivalent to `Static.Expr.t`, whose data constructors are exposed in the signature `CSS`.

In the case of `ExprBind`, it turns out that this loss of data abstraction is actually irrelevant. The data constructors of `Expr.t` are not visible in the signature of `Expr` or `X.Expr`, only in the signature of `Static.Expr`. Since `Static` is an internal, elaborator-generated module variable name, the implementation of `Bind` has no way of projecting from `Static` directly. Consequently, while `Bind` gets to "know" that `Expr.t` is implemented as a `datatype`, it has no way of exploiting this knowledge. Thus, for all intents and purposes, `Expr.t` is an abstract type. (Similarly, as far as `Expr` is concerned, `Bind.t` is effectively abstract as well.)

Unfortunately, the elaboration approach of Figure 5.22 only succeeds in enforcing data abstraction boundaries between `Expr` and `Bind` because `Expr.t` and `Bind.t` are implemented by `datatype` definitions. To illustrate this point, let us consider hypothetically how the elaboration of `ExprBind` would differ if `Expr.t` and `Bind.t` were implemented internally by the transparent `type` bindings `type t =` C and `type t =` D, respectively, instead of by `datatype` bindings. Figure 5.23 illustrates how the closed static signature and the elaborated recursive module body would change accordingly. (Note that, in order to obey the dynamic-on-static restriction, it must be the case

```
metasig
  structure Expr : {public = sig type t end,
                    private = sig type t = C end}
  structure Bind : {public = sig type t end,
                    private = sig type t = D end}
end
```

Figure 5.24: Meta-signature for Modified `ExprBind`

that neither C nor D refers to the recursive variable `X`.)

The problem with this elaboration is that the signature `CSS` reveals the identities of `Expr.t` and `Bind.t` to the implementations of *both* modules. In order for data abstraction to be enforced, `Expr` should not be able to see how `Static.Bind.t` is defined, nor should `Bind` be able to see how `Static.Expr.t` is defined. On the other hand, in order to avoid double vision, the implementation of `Expr` needs to know that `Static.Expr.t` is equivalent to C, and the implementation of `Bind` needs to know that `Static.Bind.t` is equivalent to D.

What this tells us is that the implementations of `Expr` and `Bind` really ought to be typechecked under different typing contexts. In other words, the elaborator needs to be able to actively change the signature with which `Static` is bound in the typing context, depending on what part of the recursive module body it is currently elaborating. This kind of "context switching" is not supported directly by the IL type system, so the elaboration algorithm must implement it in some *ad hoc* way.

To implement context switching, my elaboration algorithm employs a novel mechanism called a "meta-signature." Meta-signatures are a superclass of ordinary IL signatures in which the specifications of submodule components are allowed to provide both a "public" and a "private" signature. The purpose of meta-signatures is to encapsulate the type information that is known at different points during the typechecking of a recursive module.

For example, Figure 5.24 shows a meta-signature describing the knowledge of type information within the `ExprBind` module. In this meta-signature, the public signatures of `Expr` and `Bind` indicate what is publicly known about their type components, whereas the private signatures of `Expr` and `Bind` indicate what is privately known within the modules themselves. Note that the closed static signature `CSS` of `ExprBind` is obtainable from its meta-signature by ignoring the public signatures and just looking at the private ones.

The purpose of `ExprBind`'s meta-signature is to tell the elaborator how the signature of `Static` should change at different points during the typechecking of `ExprBind`. When elaborating the implementation of `Expr`, the meta-signature indicates that the signature of `Static` should use the private signature for `Expr` and the public signature for `Bind`. When elaborating the implementation of `Bind`, the private and public signatures are reversed.

Although the `ExprBind` example does not illustrate it, the meta-signature approach also allows for proper treatment of data abstraction in the presence of *nested* sealing. For example, suppose that `Expr` contained a sealed substructure `Sub` providing an abstract type component `u` implemented internally as `int`. This would be reflected in the meta-signature of `ExprBind` by having the private signature of `Expr` be itself a meta-signature:

```
metasig
  type t = C
  structure Sub : {public = sig type u end,
                   private = sig type u = int end}
end
```

Correspondingly, during the typechecking of `Expr.Sub`, the signature of `Static.Expr` would be

```
sig
  type t = C
  structure Sub : sig type u = int end
end
```

whereas during the typechecking of `Expr` outside of `Sub`, the signature of `Static.Expr` would be

```
sig
  type t = C
  structure Sub : sig type u end
end
```

This ensures that the identity of `Static.Expr.t` is known to all of `Expr`, but the identity of `Static.Expr.Sub.u` is only known to `Expr.Sub`.

Finally, it should be understood that meta-signatures are purely an elaboration mechanism. The elaborator uses them, as part of typechecking, to ensure that `Expr` and `Bind` respect each other's abstraction boundaries. Once that is verified, the elaborator throws away the meta-signature and translates `ExprBind` precisely as shown in Figure 5.23. Thus, the meta-signature technique does not require any extensions to the IL type system.

Chapter 6

Type-Theoretic Extensions for Recursive Modules

In this chapter, I extend my module type system of Chapters 3 and 4 with support for recursive type constructors, recursively dependent signatures, and recursive modules. The type system resulting from these extensions will serve as the basis of the internal language (IL) of the new ML dialect presented in Chapter 8.

As explained in Chapter 5, all of the new recursive constructs are restricted in various ways, so that their inclusion in the IL does not significantly complicate typechecking. Recursive type constructors are *iso-recursive*, meaning that the type system requires the use of explicit coercions to mediate between a recursive type and its unfolding. Recursively dependent signatures must obey a dynamic-on-static restriction, prohibiting recursive dependencies from within the specifications of type components. Recursive modules are required syntactically to have transparent declared signatures in order to avoid the double vision problem. As a result, none of the extensions presented in this chapter poses any major technical difficulties. Sections 6.1 through 6.4 present the extensions to the languages of type constructors, terms, signatures, and modules, in that order.

There are a few subtleties, however, that may be of general interest. One point of note is the subtyping rule for recursively dependent signatures, which relies on singleton signatures in an unexpected way. Another is that, in the presence of both singleton kinds and recursive type constructors of higher kind, there exist recursive types—or, more precisely, projections from recursive type constructors—whose expansions are not well-formed under the iso-recursive form of type equivalence. To address this issue, I identify a simple restriction on the *kind* of a recursive type constructor, which guarantees that its expansion will be well-formed. While this is not the weakest restriction possible, it is easy to explain and general enough to support the elaboration of all the recursive module examples from Chapter 5.

6.1 Constructor-Language Extensions

Figure 6.1 shows the extensions to the syntax of the constructor language from Chapter 3, and Figure 6.2 shows the well-formedness and equivalence rules for the new type constructors. The recursive type constructor $\mu\alpha{:}\mathrm{K.C}$ is *iso-recursive*. This means that $\mu\alpha{:}\mathrm{K.C}$ is *not* equivalent to its fixed-point expansion $\mathrm{C}[\mu\alpha{:}\mathrm{K.C}/\alpha]$. In addition, two recursive constructors $\mu\alpha{:}\mathrm{K}_1.\mathrm{C}_1$ and $\mu\alpha{:}\mathrm{K}_2.\mathrm{C}_2$ are only equivalent if the K_i are equivalent and the C_i are equivalent.

In order to coerce between a recursive constructor and its expansion, I will introduce in Sec-

$$\begin{array}{lrcl} \text{Type Constructors} & \mathrm{C}, \mathrm{D} & ::= & \cdots \mid \mu\alpha{:}\mathrm{K}.\mathrm{C} \\ \text{Base Types} & b & ::= & \cdots \mid \mathrm{C}_1 \rightsquigarrow \mathrm{C}_2 \mid \mathsf{maybe}(\mathrm{C}) \end{array}$$

Figure 6.1: Extensions to Type Constructor Syntax

$\boxed{\textbf{Well-formed constructors: } \Delta \vdash \mathrm{C} : \mathrm{K}}$

$$\frac{\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{C} : \mathrm{K}}{\Delta \vdash \mu\alpha{:}\mathrm{K}.\mathrm{C} : \mathrm{K}}\ (105) \qquad \frac{\Delta \vdash \mathrm{C}' : \mathbf{T} \quad \Delta \vdash \mathrm{C}'' : \mathbf{T}}{\Delta \vdash \mathrm{C}' \rightsquigarrow \mathrm{C}'' : \mathbf{T}}\ (106) \qquad \frac{\Delta \vdash \mathrm{C} : \mathbf{T}}{\Delta \vdash \mathsf{maybe}(\mathrm{C}) : \mathbf{T}}\ (107)$$

$\boxed{\textbf{Constructor equivalence: } \Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}}$

$$\frac{\Delta \vdash \mathrm{K}_1 \equiv \mathrm{K}_2 \quad \Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathrm{K}_1}{\Delta \vdash \mu\alpha{:}\mathrm{K}_1.\mathrm{C}_1 \equiv \mu\alpha{:}\mathrm{K}_2.\mathrm{C}_2 : \mathrm{K}_1}\ (108)$$

$$\frac{\Delta \vdash \mathrm{C}_1' \equiv \mathrm{C}_2' : \mathbf{T} \quad \Delta \vdash \mathrm{C}_1'' \equiv \mathrm{C}_2'' : \mathbf{T}}{\Delta \vdash \mathrm{C}_1' \rightsquigarrow \mathrm{C}_1'' \equiv \mathrm{C}_2' \rightsquigarrow \mathrm{C}_2'' : \mathbf{T}}\ (109) \qquad \frac{\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}}{\Delta \vdash \mathsf{maybe}(\mathrm{C}_1) \equiv \mathsf{maybe}(\mathrm{C}_2) : \mathbf{T}}\ (110)$$

Figure 6.2: Inference Rules for Type Constructors

tion 6.2 explicit fold_C and $\mathsf{unfold}_\mathrm{C}$ coercion functions. The "coercion" base type $\mathrm{C}_1 \rightsquigarrow \mathrm{C}_2$ describes the types of these functions. For instance, if we were to add sum types to the language, the type of integer lists could be represented as $\mathrm{D} = \mu\alpha{:}\mathbf{T}.(\mathsf{unit} + \mathsf{int} \times \alpha)$. In this case, fold_D would have type $(\mathsf{unit} + \mathsf{int} \times \mathrm{D}) \rightsquigarrow \mathrm{D}$, and $\mathsf{unfold}_\mathrm{D}$ would have type $\mathrm{D} \rightsquigarrow (\mathsf{unit} + \mathsf{int} \times \mathrm{D})$.

This example shows how fold_D and $\mathsf{unfold}_\mathrm{D}$ coerce between a recursive *type* and its expansion, but what about a general recursive *constructor* of higher kind? There is no way for a term-level coercion to witness the conversion of $\mathrm{D} = \mu\alpha{:}\mathrm{K}.\mathrm{C}$ to/from $\mathrm{C}[\mathrm{D}/\alpha]$ directly if $\mathrm{K} \neq \mathbf{T}$. However, what fold and unfold *can* witness is the conversion of $\mathcal{E}\{\mathrm{D}\}$ to/from $\mathcal{E}\{\mathrm{C}[\mathrm{D}/\alpha]\}$, where $\mathcal{E}$ is some elimination context that drives K down to $\mathbf{T}$. For instance, if $\mathrm{K} = \mathbf{T} \times \mathbf{T}$, in which case D defines a pair of mutually recursive types, then we can fold or unfold at either $\pi_1\mathrm{D}$ or $\pi_2\mathrm{D}$. If $\mathrm{K} = \mathbf{T} \to \mathbf{T}$, in which case D is a polymorphic recursive type, then we can fold or unfold at any instantiation of D, *i.e.,* at $\mathrm{D}(\mathrm{C}')$ for any type C' (of kind $\mathbf{T}$). The typing of fold_C and $\mathsf{unfold}_\mathrm{C}$ will be studied in Section 6.2.

While it is not necessary to distinguish the type of coercion functions from the arrow type of ordinary functions, there is a practical benefit in doing so. If values of a recursive type C are represented in a compiler in the same way as values of C's expansion, applications of fold_C and $\mathsf{unfold}_\mathrm{C}$ may be erased during code generation because they will have no run-time effect. Since fold_C and $\mathsf{unfold}_\mathrm{C}$ are the only closed values of coercion type, applications of all values of coercion type—including variables or, more generally, paths—may be erased during code generation as well.

Vanderwaart *et al.* [79] have shown that coercion types thus provide a type-preserving way of making `datatype` constructor and destructor applications more efficient. The issue is as follows. In a type-preserving compiler for ML, since `datatype`'s are opaque, a call to a `datatype` constructor defined in a separately compiled module cannot safely be inlined because it is not known what recursive type is used to implement the `datatype` (there is more than one possibility). However,

$\boxed{\textbf{Base type well-formedness: } \Delta \vdash b \text{ ok}}$

$$\frac{\Delta \vdash \mathrm{C}_1 \Leftarrow \mathbf{T} \quad \Delta \vdash \mathrm{C}_2 \Leftarrow \mathbf{T}}{\Delta \vdash \mathrm{C}_1 \rightsquigarrow \mathrm{C}_2 \text{ ok}} \qquad \frac{\Delta \vdash \mathrm{C} \Leftarrow \mathbf{T}}{\Delta \vdash \mathsf{maybe}(\mathrm{C}) \text{ ok}}$$

$\boxed{\textbf{Principal kind synthesis: } \Delta \vdash \mathrm{C} \Rightarrow \mathrm{K}}$

$$\frac{\Delta \vdash \mathrm{K} \text{ kind} \quad \Delta, \alpha{:}\mathrm{K} \vdash \mathrm{C} \Leftarrow \mathrm{K}}{\Delta \vdash \mu\alpha{:}\mathrm{K}.\mathrm{C} \Rightarrow \mathfrak{s}_{\mathrm{K}}(\mu\alpha{:}\mathrm{K}.\mathrm{C})}$$

Figure 6.3: Extensions to Kind Synthesis

Recursive Type Paths	Q	::=	$\mathcal{E}\{\mu\alpha{:}\mathrm{K}.\mathrm{C}\}$
Constructor Paths	P	::=	$\cdots \mid \mathrm{Q}$

$\boxed{\textbf{Natural kind extraction: } \Delta \vdash \mathrm{P} \uparrow \mathrm{K}}$

$\Delta \vdash \mu\alpha{:}\mathrm{K}.\mathrm{C} \uparrow \mathrm{K}$

$\boxed{\textbf{Algorithmic path equivalence: } \Delta \vdash \mathrm{P}_1 \leftrightarrow \mathrm{P}_2 \uparrow \mathrm{K}}$

$\Delta \vdash \mu\alpha{:}\mathrm{K}_1.\mathrm{C}_1 \leftrightarrow \mu\alpha{:}\mathrm{K}_2.\mathrm{C}_2 \uparrow \mathrm{K}_1$	if $\Delta \vdash \mathrm{K}_1 \Leftrightarrow \mathrm{K}_2$ and $\Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}_1 \Leftrightarrow \mathrm{C}_2 : \mathrm{K}_1$
$\Delta \vdash \mathrm{C}_1' \rightsquigarrow \mathrm{C}_1'' \leftrightarrow \mathrm{C}_2' \rightsquigarrow \mathrm{C}_2'' \uparrow \mathbf{T}$	if $\Delta \vdash \mathrm{C}_1' \Leftrightarrow \mathrm{C}_2' : \mathbf{T}$ and $\Delta \vdash \mathrm{C}_1'' \Leftrightarrow \mathrm{C}_2'' : \mathbf{T}$
$\Delta \vdash \mathsf{maybe}(\mathrm{C}_1) \leftrightarrow \mathsf{maybe}(\mathrm{C}_2) \uparrow \mathbf{T}$	if $\Delta \vdash \mathrm{C}_1 \Leftrightarrow \mathrm{C}_2 : \mathbf{T}$

Figure 6.4: Extensions to Constructor Equivalence Algorithm

if the `datatype`'s constructor and destructor have coercion type, then the compiler may eliminate all calls to them during code generation. This saves `datatype` constructor and destructor applications from incurring the overhead of a function call. My language definition in Chapter 9 adopts Vanderwaart *et al.*'s approach in its interpretation of `datatype`'s.

The base type $\mathsf{maybe}(\mathrm{C})$ describes memory locations that may or may not contain a value of type C. This is the type analogue of the signature $\mathsf{maybe}(\mathrm{S})$, whose utility was discussed in Section 5.2.4, and which will be formally added to the language in Section 6.3. The type $\mathsf{maybe}(\mathrm{C})$ is needed so that we can phase-split the signature $\mathsf{maybe}(\mathrm{S})$ into the core language.

Figures 6.3 and 6.4 show the extensions to the kind synthesis and constructor equivalence algorithms. Both extensions are completely straightforward. It is worth noting, though, that the recursive type constructor introduces a new irreducible constructor path of the form $\mathcal{E}\{\mu\alpha{:}\mathrm{K}.\mathrm{C}\}$. I call this a "recursive type path" and denote it with the metavariable Q. While $\mathrm{Q} = \mathcal{E}\{\mu\alpha{:}\mathrm{K}.\mathrm{C}\}$ is potentially a WHNF, it is not *necessarily* one. For example, if $\mathrm{K} = \Sigma\alpha{:}\mathbf{T}.\mathfrak{s}(\alpha \times \alpha)$, then $\pi_2\mathrm{Q}$ will not be a WHNF, since the natural kind of $\pi_2\mathrm{Q}$ will be $\mathfrak{s}(\pi_1\mathrm{Q} \times \pi_1\mathrm{Q})$. Rather, $\pi_2\mathrm{Q}$ will reduce to $\pi_1\mathrm{Q} \times \pi_1\mathrm{Q}$.

I omit proof that these extensions do not disturb the Stone-Harper meta-theory, but I have verified as much on paper. The proof is easy, primarily because we have not made any changes to the kind structure. The only interesting aspect of the extension is the μ-constructor. The iso-recursive nature of μ-equivalence makes the extension tantamount to adding an infinite set of

$$\begin{array}{lrl} \text{Values} & v ::= & \cdots \mid \mathsf{fold}_{\mathrm{C}} \mid \mathsf{unfold}_{\mathrm{C}} \mid \mathsf{fold}_{\mathrm{C}}\langle\!\langle v \rangle\!\rangle \\ \text{Terms} & e ::= & v_1\langle\!\langle v_2 \rangle\!\rangle \mid \mathsf{fetch}(v) \mid \mathsf{rec}(x : \mathrm{C}.\, e) \end{array}$$

$$\begin{array}{lcl} e_1\langle\!\langle e_2 \rangle\!\rangle & \stackrel{\text{def}}{=} & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ x_1\langle\!\langle x_2 \rangle\!\rangle \\ \mathsf{fetch}(e) & \stackrel{\text{def}}{=} & \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{fetch}(x) \end{array}$$

Figure 6.5: Extensions to Term Syntax

$$\mathsf{expand}(\mathrm{Q}) \stackrel{\text{def}}{=} \mathcal{E}\{\mathrm{C}[\mu\alpha{:}\mathrm{K.C}/\alpha]\}, \quad \text{where } \mathrm{Q} = \mathcal{E}\{\mu\alpha{:}\mathrm{K.C}\}$$

$\boxed{\textbf{Well-formed terms: } \Gamma \vdash e : \mathrm{C}}$

$$\frac{\Gamma \vdash \mathrm{C} \equiv \mathrm{Q} : \mathbf{T} \quad \Gamma \vdash \mathrm{Q}\ \mathsf{expands}}{\Gamma \vdash \mathsf{fold}_{\mathrm{C}} : \mathsf{expand}(\mathrm{Q}) \rightsquigarrow \mathrm{Q}}\ (111) \qquad \frac{\Gamma \vdash \mathrm{C} \equiv \mathrm{Q} : \mathbf{T} \quad \Gamma \vdash \mathrm{Q}\ \mathsf{expands}}{\Gamma \vdash \mathsf{unfold}_{\mathrm{C}} : \mathrm{Q} \rightsquigarrow \mathsf{expand}(\mathrm{Q})}\ (112)$$

$$\frac{\Gamma \vdash v : \mathrm{C}' \rightsquigarrow \mathrm{C} \quad \Gamma \vdash v' : \mathrm{C}'}{\Gamma \vdash v\langle\!\langle v' \rangle\!\rangle : \mathrm{C}}\ (113) \qquad \frac{\Gamma \vdash v : \mathsf{maybe}(\mathrm{C})}{\Gamma \vdash \mathsf{fetch}(v) : \mathrm{C}}\ (114) \qquad \frac{\Gamma, x{:}\mathsf{maybe}(\mathrm{C}) \vdash e : \mathrm{C}}{\Gamma \vdash \mathsf{rec}(x : \mathrm{C}.\, e) : \mathrm{C}}\ (115)$$

Figure 6.6: New Inference Rules for Terms

primitive constants μ_{K} to the language, where μ_{K} has kind $(\mathrm{K} \to \mathrm{K}) \to \mathrm{K}$, and $\mu_{\mathrm{K}}(\mathrm{C})$ is equivalent to $\mu_{\mathrm{K}}(\mathrm{D})$ if and only if C is equivalent to D at $\mathrm{K} \to \mathrm{K}$. Extending the language with constructor constants is not problematic—they behave just like variables. Given such an extension, $\mu\alpha{:}\mathrm{K.C}$ may be viewed as a more concise way of writing $\mu_{\mathrm{K}}(\lambda\alpha{:}\mathrm{K.C})$.

6.2 Term-Language Extensions

Figure 6.5 shows the extensions to the syntax of terms. New term forms include the $\mathsf{fold}_{\mathrm{C}}$ and $\mathsf{unfold}_{\mathrm{C}}$ coercion values and the coercion application $v_1\langle\!\langle v_2 \rangle\!\rangle$. The application of $\mathsf{fold}_{\mathrm{C}}$ to a value is also considered a value—in fact, $\mathsf{fold}_{\mathrm{C}}\langle\!\langle v \rangle\!\rangle$ is the canonical form inhabiting recursive types Q. The terms $\mathsf{fetch}(v)$ and $\mathsf{rec}(x : \mathrm{C}.\, e)$ are the term analogues of the module constructs $\mathsf{fetch}(\mathrm{M})$ and $\mathsf{rec}(\mathrm{X} : \mathrm{S}.\, \mathrm{M})$, which will be introduced in Section 6.4. The former dereferences the memory location represented by v, raising a (run-time) error if v's contents are undefined; the latter allocates a fresh location x of type $\mathsf{maybe}(\mathrm{C})$, evaluates e to a value v, and then backpatches x with v.

Figure 6.6 gives the typing rules for these new term constructs. Rules 113–115 are straightforward. Rules 111 and 112 for the $\mathsf{fold}_{\mathrm{C}}$ and $\mathsf{unfold}_{\mathrm{C}}$ values make use of a new macro, $\mathsf{expand}(\mathrm{Q})$, and a new judgment, $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$. First, $\mathsf{expand}(\mathrm{Q})$ is the constructor produced by replacing Q's head, which has the form $\mu\alpha{:}\mathrm{K.C}$, with its fixed-point expansion $\mathrm{C}[\mu\alpha{:}\mathrm{K.C}/\alpha]$. If C is equivalent to a recursive type path Q, then $\mathsf{fold}_{\mathrm{C}}$ will be a coercion function from $\mathsf{expand}(\mathrm{Q})$ to Q, and $\mathsf{unfold}_{\mathrm{C}}$ will coerce in the opposite direction.

The judgment $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$, invoked in the second premise of Rules 111 and 112, is defined in Figure 6.7. The purpose of this premise is to ensure (1) that $\mathsf{fold}_{\mathrm{C}}$ and $\mathsf{unfold}_{\mathrm{C}}$ have unique types, and (2) that their types are well-formed. I will demonstrate first how both of these conditions may

Singleton-Free Kinds	$k, \ell ::= 1 \mid \mathbf{T} \mid k_1 \times k_2 \mid k_1 \rightarrow k_2$
Expandable Kinds	$\mathcal{K}, \mathcal{L} ::= \mathbb{K} \mid \mathbf{T} \mid \Sigma\alpha{:}\mathcal{K}_1.\mathcal{K}_2 \mid \Pi\alpha{:}k_1.\mathcal{K}_2$

$\boxed{\textbf{Expandable recursive types: } \Delta \vdash \mathrm{Q}\ \mathsf{expands}}$

$$\frac{\Delta \vdash \mathrm{Q} : \mathbf{T} \quad \Delta \vdash \mathrm{Q} \uparrow \mathbf{T} \quad \mathrm{Q} = \mathcal{E}\{\mu\alpha{:}\mathcal{K}.\mathrm{C}\}}{\Delta \vdash \mathrm{Q}\ \mathsf{expands}}$$

Figure 6.7: Expandable Kinds and Types

$\boxed{\textbf{Type synthesis: } \Gamma \vdash e \Rightarrow \mathrm{C}}$

$$\frac{\Gamma \vdash \mathrm{C} : \mathbf{T} \quad \Gamma \vdash \mathrm{C} \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{Q} \quad \Gamma \vdash \mathrm{Q}\ \mathsf{expands}}{\Gamma \vdash \mathsf{fold}_{\mathrm{C}} \Rightarrow \mathsf{expand}(\mathrm{Q}) \rightsquigarrow \mathrm{Q}} \qquad \frac{\Gamma \vdash \mathrm{C} : \mathbf{T} \quad \Gamma \vdash \mathrm{C} \stackrel{\mathrm{wh}}{\Longrightarrow} \mathrm{Q} \quad \Gamma \vdash \mathrm{Q}\ \mathsf{expands}}{\Gamma \vdash \mathsf{unfold}_{\mathrm{C}} \Rightarrow \mathrm{Q} \rightsquigarrow \mathsf{expand}(\mathrm{Q})}$$

$$\frac{\Gamma \vdash v \Rightarrow \mathrm{C}' \rightsquigarrow \mathrm{C} \quad \Gamma \vdash v' \Leftarrow \mathrm{C}'}{\Gamma \vdash v\langle\!\langle v' \rangle\!\rangle \Rightarrow \mathrm{C}} \qquad \frac{\Gamma \vdash v \Rightarrow \mathsf{maybe}(\mathrm{C})}{\Gamma \vdash \mathsf{fetch}(v) \Rightarrow \mathrm{C}} \qquad \frac{\Gamma \vdash \mathrm{C} : \mathbf{T} \quad \Gamma, x{:}\mathsf{maybe}(\mathrm{C}) \vdash e \Leftarrow \mathrm{C}}{\Gamma \vdash \mathsf{rec}(x : \mathrm{C}.\, e) \Rightarrow \mathrm{C}}$$

Figure 6.8: Extensions to Type Synthesis

fail in the absence of this premise, and second how the definition of $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$ in Figure 6.7 ensures that they do not.

Suppose that the second premise of Rules 111 and 112 were omitted. The first problem is that, by singleton reasoning, *every* type C is equivalent to a recursive type Q, namely the type $\mathrm{Q} = \mu\alpha{:}\mathfrak{S}(\mathrm{C}).\mathrm{C}$. In this case, $\mathsf{expand}(\mathrm{Q})$ is precisely C, so $\mathsf{fold}_{\mathrm{C}}$ and $\mathsf{unfold}_{\mathrm{C}}$ could always be given the type $\mathrm{C} \rightsquigarrow \mathrm{C}$. Of course, C might be equivalent to a "real" recursive type, such as $\mu\beta{:}\mathbf{T}.\mathrm{D}$, in which case $\mathsf{fold}_{\mathrm{C}}$ could also be given the type $\mathrm{D}[\mathrm{C}/\beta] \rightsquigarrow \mathrm{C}$. Since C is not equivalent to $\mathrm{D}[\mathrm{C}/\beta]$, we would therefore lose unicity of types.

The second problem is that the well-formedness of Q does not imply the well-formedness of $\mathsf{expand}(\mathrm{Q})$. For example, consider the following set of contrived, yet illustrative, definitions:

$$\begin{array}{lcl} \mathrm{K} & \stackrel{\mathrm{def}}{=} & \Sigma\beta{:}\mathbf{T}.\mathfrak{S}(\beta) \rightarrow \mathbf{T} \\ \mathrm{C} & \stackrel{\mathrm{def}}{=} & \mu\alpha{:}\mathrm{K}.\langle \mathsf{unit}, \lambda\alpha'{:}\mathfrak{S}(\mathsf{unit}).\mathsf{unit}\rangle \\ \mathrm{Q} & \stackrel{\mathrm{def}}{=} & (\pi_2\mathrm{C})(\pi_1\mathrm{C}) \end{array}$$

Note that Q has the form $\mathcal{E}\{\mathrm{C}\}$, where $\mathcal{E} = (\pi_2(\bullet))(\pi_1\mathrm{C})$. As C has kind K, it is clear that Q is a well-formed type. However, $\mathsf{expand}(\mathrm{Q}) = (\pi_2(\mathsf{expand}(\mathrm{C})))(\pi_1\mathrm{C})$ is not well-formed. Specifically, $\pi_2(\mathsf{expand}(\mathrm{C}))$ has arrow kind $\mathfrak{S}(\pi_1(\mathsf{expand}(\mathrm{C}))) \rightarrow \mathbf{T}$, whose argument kind is not matched by $\pi_1\mathrm{C}$. This problem does not arise under an equi-recursive account of constructor equivalence—in that setting, C would be considered equivalent to $\mathsf{expand}(\mathrm{C})$, so $\mathsf{expand}(\mathrm{Q})$ would be well-formed. Of course, in an equi-recursive setting, $\mathsf{fold}_{\mathrm{C}}$ and $\mathsf{unfold}_{\mathrm{C}}$ are not necessary in the first place.

The judgment, $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$, addresses the unicity-of-types problem by requiring Q to be in weak head normal form. Formally, this means that Q is a type path rooted at a μ-constructor and its natural kind is $\mathbf{T}$. This restriction prevents one from passing off any type as a recursive type, since $\mu\alpha{:}\mathfrak{S}(\mathrm{C}).\mathrm{C}$ is not in WHNF (its natural kind is $\mathfrak{S}(\mathrm{C})$, so it reduces to C). Only types whose WHNF's have the form Q may be used as the parameter to fold and unfold. As this marks the first intrusion of natural kind extraction into the core language *type system*, it is important to show that natural kind extraction for recursive type paths obeys weakening and substitution. In fact, this turns out to be trivial, since the procedure for extracting the natural kind of a recursive type path never even inspects the typing context! Only paths rooted at *variables* require inspection of the typing context.

Proposition 6.2.1 (Natural Kind Extraction for Recursive Type Paths)
If $\Delta \vdash \mathrm{Q} \uparrow \mathrm{K}$, then $\Delta' \vdash \gamma\mathrm{Q} \uparrow \gamma\mathrm{K}$ for any Δ' and γ.

As for the second problem, the most obvious solution would be for the judgment $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$ to require that $\Delta \vdash \mathsf{expand}(\mathrm{Q}) : \mathbf{T}$. This is clearly a necessary condition. While I conjecture that it is also a sufficient one, I have been unable to prove that the resulting type checking algorithm (Figure 6.8) is complete. In particular, the completeness proof relies on the property, stated below in Theorem 6.2.5, that if Q and Q′ are equivalent WHNF types and $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$, then $\mathsf{expand}(\mathrm{Q}')$ is well-formed as well and equivalent to $\mathsf{expand}(\mathrm{Q})$. In order to prove this theorem, I impose a stronger condition on $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$ than the well-formedness of $\mathsf{expand}(\mathrm{Q})$—I require that Q be syntactically *expandable*.

Figure 6.7 defines a recursive type path Q to be expandable if Q's head has the form $\mu\alpha{:}\mathcal{K}.\mathrm{C}$, where $\mathcal{K}$ is a syntactically expandable kind. The definition of expandable kinds is motivated conversely by the desire to ensure that all expandable constructors have well-formed expansions, *i.e.,* that $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$ implies $\Delta \vdash \mathsf{expand}(\mathrm{Q}) : \mathbf{T}$. Toward this end, the only restriction placed on an expandable kind $\mathcal{K}$ is that, for any arrow kind within $\mathcal{K}$ of the form $\Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2$, either the result kind K_2 must be transparent or the argument kind K_1 must be singleton-free. This avoids precisely the sort of counterexample given above, in which the kind K of the μ-constructor took the form $\Sigma\beta{:}\mathbf{T}.\mathfrak{S}(\beta) \to \mathbf{T}$. In essence, it is a "monster-barring" restriction (cf. Lakatos [40, 27]).

Intuitively, the reason the restriction works is that it prevents the well-formedness of an expandable type $\mathrm{Q} = \mathcal{E}\{\mu\alpha{:}\mathcal{K}.\mathrm{C}\}$ from depending on the fact that its head is $\mu\alpha{:}\mathcal{K}.\mathrm{C}$ as opposed to any other constructor of kind $\mathcal{K}$. That is, if Q is well-formed, then $\mathcal{E}\{\mathrm{D}\}$ will be well-formed for *any* D that has kind $\mathcal{K}$. This includes $\mathsf{expand}(\mathrm{Q}) = \mathcal{E}\{\mathrm{C}[\mu\alpha{:}\mathcal{K}.\mathrm{C}/\alpha]\}$.

While the syntactic restriction on expandable kinds is more conservative than absolutely necessary, it has the advantage of being easy to define and understand.[1] Furthermore, it enables us to support the elaboration of all the recursive module examples from Chapter 5. The only sorts of recursive modules whose elaboration it prevents are those that contain `datatype` definitions in the bodies of `functor` bindings. As I have not yet seen any compelling examples of recursive functors, I suspect that this is not a great loss. (See Section 9.3.8 for more detailed discussion of this issue.)

I will now proceed to prove the critical Theorem 6.2.5. The proof relies on a technical "head replacement" lemma (Lemma 6.2.4), whose statement depends on a notion of structural similarity of kinds. Two kinds are structurally similar if they appear identical when one ignores the contents of their constituent singleton kinds. Here is the formal definition, followed by a proposition enumerating several properties of structural similarity that are provable by straightforward induction.

[1] In addition, as I expect in future work to be able to eliminate a syntactic condition on Q altogether and replace it with the minimal requirement that $\mathsf{expand}(\mathrm{Q})$ be well-formed, I see little point in developing finer, more complex approximations of expandability, only to discard them in the near future.

Definition 6.2.2 (Structural Similarity of Kinds)
Kinds K_1 and K_2 are *structurally similar* (written $K_1 \approx K_2$) if:

- $K_1 = K_2$
- Or, $K_1 = \mathfrak{S}(C_1)$ and $K_2 = \mathfrak{S}(C_2)$
- Or, $K_1 = \Sigma\alpha{:}K_1'.K_1''$ and $K_2 = \Sigma\alpha{:}K_2'.K_2''$ and $K_1' \approx K_2'$ and $K_1'' \approx K_2''$
- Or, $K_1 = \Pi\alpha{:}K_1'.K_1''$ and $K_2 = \Pi\alpha{:}K_2'.K_2''$ and $K_1' \approx K_2'$ and $K_1'' \approx K_2''$

Proposition 6.2.3 (Properties of Structural Similarity)

1. Structural similarity is an equivalence relation.
2. If $K_1 \approx K_2$ and K_1 is singleton-free, then $K_1 = K_2$.
3. If $K_1 \approx K_2$ and K_1 is expandable, then K_2 is expandable.
4. If $K_1 \approx K_2$, then $\gamma_1 K_1 \approx \gamma_2 K_2$ for any γ_1 and γ_2.
5. If $\Delta \vdash K_1 \equiv K_2$, then $K_1 \approx K_2$.

The head replacement lemma has two parts. The first part says that if $\mathcal{E}\{P\}$ is a well-formed type in WHNF, and if P has an expandable natural kind $\mathcal{L}$, then we can replace P by any other path P' of kind $\mathcal{L}$, and the resulting type $\mathcal{E}\{P'\}$ will be well-formed. The second part says that if $\mathcal{E}_1\{P_1\}$ and $\mathcal{E}_2\{P_2\}$ are algorithmically equivalent paths of kind $\mathbf{T}$, and the natural kind of P_1 is the expandable $\mathcal{L}$, then we can replace the P_i by any other pair of algorithmically equivalent constructors P_i' of kind $\mathcal{L}$, and the resulting $\mathcal{E}_i\{P_i'\}$ will be algorithmically equivalent as well. The lemma is stated in a somewhat more general fashion in order to make the induction go through.

Lemma 6.2.4 (Head Replacement)

1. Suppose $\Delta \vdash P \uparrow \mathcal{L}$ and $\Delta \vdash P' : \mathcal{L}'$, where $\mathcal{L} \approx \mathcal{L}'$.
   If $\mathcal{E}\{P\}$ is well-formed in Δ and $\Delta \vdash \mathcal{E}\{P\} \uparrow K$, where K is not transparent,
   then there exists $\mathcal{K}'$ such that $\Delta \vdash \mathcal{E}\{P'\} : \mathcal{K}'$ and $K \approx \mathcal{K}'$.
2. Suppose $\Delta \vdash P_1 \leftrightarrow P_2 \uparrow \mathcal{L}$ and $\Delta \vdash P_1' \leftrightarrow P_2' \uparrow \mathcal{L}'$, where $\mathcal{L} \approx \mathcal{L}'$.
   If $\Delta \vdash \mathcal{E}_1\{P_1\} \leftrightarrow \mathcal{E}_2\{P_2\} \uparrow K$, where K is not transparent,
   then there exists $\mathcal{K}'$ such that $\Delta \vdash \mathcal{E}_1\{P_1'\} \leftrightarrow \mathcal{E}_2\{P_2'\} \uparrow \mathcal{K}'$ and $K \approx \mathcal{K}'$.

Proof: By induction on the structure of $\mathcal{E}$ in Part 1, and $\mathcal{E}_1$ and $\mathcal{E}_2$ in Part 2.

1.
   - Case: $\mathcal{E} = \bullet$. Trivial.
   - Case: $\mathcal{E} = \pi_1\mathcal{E}'$.
     (a) We have $\Delta \vdash \mathcal{E}\{P\} \uparrow K_1$, where $\Delta \vdash \mathcal{E}'\{P\} \uparrow \Sigma\alpha{:}K_1.K_2$.
     (b) By induction, $\Delta \vdash \mathcal{E}'\{P'\} : \Sigma\alpha{:}\mathcal{K}_1'.\mathcal{K}_2'$, where $K_1 \approx \mathcal{K}_1'$ and $K_2 \approx \mathcal{K}_2'$.
     (c) Thus, $\Delta \vdash \mathcal{E}\{P'\} : \mathcal{K}_1'$.
   - Case: $\mathcal{E} = \pi_2\mathcal{E}'$.
     (a) We have $\Delta \vdash \mathcal{E}\{P\} \uparrow K_2[\pi_1\mathcal{E}'\{P\}/\alpha]$, where $\Delta \vdash \mathcal{E}'\{P\} \uparrow \Sigma\alpha{:}K_1.K_2$.
     (b) By induction, $\Delta \vdash \mathcal{E}'\{P'\} : \Sigma\alpha{:}\mathcal{K}_1'.\mathcal{K}_2'$, where $K_1 \approx \mathcal{K}_1'$ and $K_2 \approx \mathcal{K}_2'$.
     (c) Thus, $\Delta \vdash \mathcal{E}\{P'\} : \mathcal{K}_2'[\pi_1\mathcal{E}'\{P'\}/\alpha]$.
     (d) By Proposition 6.2.3, $K_2[\pi_1\mathcal{E}'\{P\}/\alpha] \approx \mathcal{K}_2'[\pi_1\mathcal{E}'\{P'\}/\alpha]$.

- Case: $\mathcal{E} = \mathcal{E}'(\mathrm{C})$.
  (a) We have $\Delta \vdash \mathcal{E}\{\mathrm{P}\} \uparrow \mathrm{K}_2[\mathrm{C}/\alpha]$, where $\Delta \vdash \mathcal{E}'\{\mathrm{P}\} \uparrow \Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2$.
  (b) Since $\mathcal{E}\{\mathrm{P}\}$ is well-formed in Δ, by Proposition 3.1.17, $\Delta \vdash \mathrm{C} : \mathrm{K}_1$.
  (c) By induction, $\Delta \vdash \mathcal{E}'\{\mathrm{P}'\} \uparrow \Pi\alpha{:}k_1'.\mathcal{K}_2'$, where $\mathrm{K}_1 \approx k_1'$ and $\mathrm{K}_2 \approx \mathcal{K}_2'$.
  (d) By Proposition 6.2.3, $\mathrm{K}_1 = k_1'$, so $\Delta \vdash \mathrm{C} : k_1'$.
  (e) Thus, $\Delta \vdash \mathcal{E}\{\mathrm{P}'\} : \mathcal{K}_2'[\mathrm{C}/\alpha]$.
  (f) By Proposition 6.2.3, $\mathrm{K}_2[\mathrm{C}/\alpha] \approx \mathcal{K}_2'[\mathrm{C}/\alpha]$.

2. - Case: $\mathcal{E}_i = \bullet$. Trivial.
   - Case: $\mathcal{E}_i = \pi_1\mathcal{E}_i'$.
     (a) We have $\Delta \vdash \mathcal{E}_1\{\mathrm{P}_1\} \leftrightarrow \mathcal{E}_2\{\mathrm{P}_2\} \uparrow \mathrm{K}_1$, where $\Delta \vdash \mathcal{E}_1'\{\mathrm{P}_1\} \leftrightarrow \mathcal{E}_2'\{\mathrm{P}_2\} \uparrow \Sigma\alpha{:}\mathrm{K}_1.\mathrm{K}_2$.
     (b) By induction, $\Delta \vdash \mathcal{E}_1'\{\mathrm{P}_1'\} \leftrightarrow \mathcal{E}_2'\{\mathrm{P}_2'\} \uparrow \Sigma\alpha{:}\mathcal{K}_1'.\mathcal{K}_2'$, where $\mathrm{K}_1 \approx \mathcal{K}_1'$ and $\mathrm{K}_2 \approx \mathcal{K}_2'$.
     (c) Thus, $\Delta \vdash \mathcal{E}_1\{\mathrm{P}_1'\} \leftrightarrow \mathcal{E}_2\{\mathrm{P}_2'\} \uparrow \mathcal{K}_1'$.
   - Case: $\mathcal{E}_i = \pi_2\mathcal{E}_i'$.
     (a) We have $\Delta \vdash \mathcal{E}_1\{\mathrm{P}_1\} \leftrightarrow \mathcal{E}_2\{\mathrm{P}_2\} \uparrow \mathrm{K}_2[\pi_1\mathcal{E}_1'\{\mathrm{P}_1\}/\alpha]$,
     where $\Delta \vdash \mathcal{E}_1'\{\mathrm{P}_1\} \leftrightarrow \mathcal{E}_2'\{\mathrm{P}_2\} \uparrow \Sigma\alpha{:}\mathrm{K}_1.\mathrm{K}_2$.
     (b) By induction, $\Delta \vdash \mathcal{E}_1'\{\mathrm{P}_1'\} \leftrightarrow \mathcal{E}_2'\{\mathrm{P}_2'\} \uparrow \Sigma\alpha{:}\mathcal{K}_1'.\mathcal{K}_2'$, where $\mathrm{K}_1 \approx \mathcal{K}_1'$ and $\mathrm{K}_2 \approx \mathcal{K}_2'$.
     (c) Thus, $\Delta \vdash \mathcal{E}_1\{\mathrm{P}_1'\} \leftrightarrow \mathcal{E}_2\{\mathrm{P}_2'\} \uparrow \mathcal{K}_2'[\pi_1\mathcal{E}_1'\{\mathrm{P}_1'\}/\alpha]$.
     (d) By Proposition 6.2.3, $\mathrm{K}_2[\pi_1\mathcal{E}_1'\{\mathrm{P}_1\}/\alpha] \approx \mathcal{K}_2'[\pi_1\mathcal{E}_1'\{\mathrm{P}_1'\}/\alpha]$.
   - Case: $\mathcal{E}_i = \mathcal{E}_i'(\mathrm{C}_i)$.
     (a) We have $\Delta \vdash \mathcal{E}_1\{\mathrm{P}_1\} \leftrightarrow \mathcal{E}_2\{\mathrm{P}_2\} \uparrow \mathrm{K}_2[\mathrm{C}_1/\alpha]$,
     where $\Delta \vdash \mathcal{E}_1'\{\mathrm{P}_1\} \leftrightarrow \mathcal{E}_2'\{\mathrm{P}_2\} \uparrow \Pi\alpha{:}\mathrm{K}_1.\mathrm{K}_2$ and $\Delta \vdash \mathrm{C}_1 \Leftrightarrow \mathrm{C}_2 : \mathrm{K}_1$.
     (b) By induction, $\Delta \vdash \mathcal{E}_1'\{\mathrm{P}_1'\} \leftrightarrow \mathcal{E}_2'\{\mathrm{P}_2'\} \uparrow \Pi\alpha{:}k_1'.\mathcal{K}_2'$, where $\mathrm{K}_1 \approx k_1'$ and $\mathrm{K}_2 \approx \mathcal{K}_2'$.
     (c) By Proposition 6.2.3, $\mathrm{K}_1 = k_1'$, so $\Delta \vdash \mathrm{C}_1 \Leftrightarrow \mathrm{C}_2 : k_1'$.
     (d) Thus, $\Delta \vdash \mathcal{E}_1\{\mathrm{P}_1'\} \leftrightarrow \mathcal{E}_2\{\mathrm{P}_2'\} \uparrow \mathcal{K}_2'[\mathrm{C}_1/\alpha]$.
     (e) By Proposition 6.2.3, $\mathrm{K}_2[\mathrm{C}_1/\alpha] \approx \mathcal{K}_2'[\mathrm{C}_1/\alpha]$.

■

Theorem 6.2.5 (Equivalent Expandable Types Have Equivalent Expansions)
If $\Delta \vdash \mathrm{Q} \equiv \mathrm{Q}' : \mathbf{T}$ and $\Delta \vdash \mathrm{Q}$ expands and $\Delta \vdash \mathrm{Q}' \uparrow \mathbf{T}$,
then $\Delta \vdash \mathrm{Q}'$ expands and $\Delta \vdash \mathsf{expand}(\mathrm{Q}) \equiv \mathsf{expand}(\mathrm{Q}') : \mathbf{T}$.

Proof:

1. Let $\mathrm{Q} = \mathcal{E}\{\mu\alpha{:}\mathrm{K}.\mathrm{C}\}$ and $\mathrm{Q}' = \mathcal{E}'\{\mu\alpha{:}\mathrm{K}'.\mathrm{C}'\}$.
2. By the equivalence algorithm, $\Delta \vdash \mu\alpha{:}\mathrm{K}.\mathrm{C} \equiv \mu\alpha{:}\mathrm{K}'.\mathrm{C}' : \mathrm{K}$,
   $\Delta \vdash \mathrm{K} \equiv \mathrm{K}'$ and $\Delta, \alpha{:}\mathrm{K} \vdash \mathrm{C} \equiv \mathrm{C}' : \mathrm{K}$.
3. By functionality, $\Delta \vdash \mathrm{C}[\mu\alpha{:}\mathrm{K}.\mathrm{C}/\alpha] \equiv \mathrm{C}'[\mu\alpha{:}\mathrm{K}'.\mathrm{C}'/\alpha] : \mathrm{K}$.
4. Since $\Delta \vdash \mathrm{Q}$ expands, K is expandable. By Proposition 6.2.3, K′ is expandable and $\mathrm{K} \approx \mathrm{K}'$.
5. Thus, $\Delta \vdash \mathrm{Q}'$ expands.
6. Let $\Delta' = \Delta, \beta{:}\mathrm{K}$. We have $\Delta' \vdash \beta : \mathrm{K}$, $\Delta' \vdash \mu\alpha{:}\mathrm{K}.\mathrm{C} \uparrow \mathrm{K}$ and $\Delta' \vdash \mu\alpha{:}\mathrm{K}'.\mathrm{C}' \uparrow \mathrm{K}'$.

7. By Part 1 of Lemma 6.2.4, $\Delta' \vdash \mathcal{E}\{\beta\} : \mathbf{T}$ and $\Delta' \vdash \mathcal{E}'\{\beta\} : \mathbf{T}$.
8. By the equivalence algorithm, $\Delta' \vdash \mathrm{Q} \leftrightarrow \mathrm{Q}' \uparrow \mathbf{T}$, $\Delta' \vdash \beta \leftrightarrow \beta \uparrow \mathrm{K}$, and $\Delta' \vdash \mu\alpha{:}\mathrm{K}.\mathrm{C} \leftrightarrow \mu\alpha{:}\mathrm{K}'.\mathrm{C}' \uparrow \mathrm{K}$.
9. By Part 2 of Lemma 6.2.4, $\Delta' \vdash \mathcal{E}\{\beta\} \leftrightarrow \mathcal{E}'\{\beta\} \uparrow \mathbf{T}$.
10. By soundness of the equivalence algorithm, $\Delta' \vdash \mathcal{E}\{\beta\} \equiv \mathcal{E}'\{\beta\} : \mathbf{T}$.
11. By functionality, $\Delta \vdash \mathcal{E}\{\mathrm{C}[\mu\alpha{:}\mathrm{K}.\mathrm{C}/\alpha]\} \equiv \mathcal{E}'\{\mathrm{C}'[\mu\alpha{:}\mathrm{K}'.\mathrm{C}'/\alpha]\} : \mathbf{T}$.
12. In other words, $\Delta \vdash \mathsf{expand}(\mathrm{Q}) \equiv \mathsf{expand}(\mathrm{Q}') : \mathbf{T}$.

■

Corollary 6.2.6 (Well-Formed Expandable Types Have Well-Formed Expansions)
If $\Delta \vdash \mathrm{Q}\ \mathsf{expands}$, then $\Delta \vdash \mathsf{expand}(\mathrm{Q}) : \mathbf{T}$.

Proof: By Theorem 6.2.5, Reflexivity, and Validity. ■

Given Theorem 6.2.5, it is easy to prove that the type synthesis algorithm, the new cases of which are shown in Figure 6.8, is complete.

Proposition 3.2.6 (Completeness of Type Checking)
If $\Gamma \vdash e : \mathrm{C}$, then $\Gamma \vdash e \Leftarrow \mathrm{C}$.

Proof: Interesting new cases: Rules 111 and 112. I will show the former, the latter is analogous.

1. We have $\Gamma \vdash \mathsf{fold}_{\mathrm{C}} : \mathsf{expand}(\mathrm{C}) \rightsquigarrow \mathrm{C}$, where $\Gamma \vdash \mathrm{C} \equiv \mathrm{Q} : \mathbf{T}$ and $\Gamma \vdash \mathrm{Q}\ \mathsf{expands}$.
2. By Proposition 3.1.19, $\Gamma \vdash \mathrm{C} \stackrel{\text{wh}}{\Longrightarrow} \mathrm{P}$ and $\Gamma \vdash \mathrm{C} \equiv \mathrm{P} : \mathbf{T}$.
3. Thus, $\Gamma \vdash \mathrm{Q} \equiv \mathrm{P} : \mathbf{T}$, $\Gamma \vdash \mathrm{Q} \uparrow \mathbf{T}$ and $\Gamma \vdash \mathrm{P} \uparrow \mathbf{T}$.
4. By the equivalence algorithm, $\Gamma \vdash \mathrm{Q} \leftrightarrow \mathrm{P} \uparrow \mathbf{T}$, so P must have the form Q'.
5. By Theorem 6.2.5, $\Gamma \vdash \mathrm{Q}'\ \mathsf{expands}$ and $\Gamma \vdash \mathsf{expand}(\mathrm{Q}) \equiv \mathsf{expand}(\mathrm{Q}') : \mathbf{T}$.
6. Thus, $\Gamma \vdash \mathsf{fold}_{\mathrm{C}} \Rightarrow \mathsf{expand}(\mathrm{Q}') \rightsquigarrow \mathrm{Q}'$, and $\Gamma \vdash \mathsf{expand}(\mathrm{Q}') \rightsquigarrow \mathrm{Q}' \equiv \mathsf{expand}(\mathrm{Q}) \rightsquigarrow \mathrm{Q} : \mathbf{T}$.

■

In order to formalize the evaluation of the recursive term construct $\mathsf{rec}(x : \mathrm{C}.\, e)$, it is useful to generalize the structural operational semantics of Section 3.2.5 to an abstract machine semantics with an explicit store and control stack.[2] Figure 6.9 illustrates how this is done.

Machine stores ω are mappings from a finite set of variables x (representing memory locations) to the contents stored at those locations. The contents of location x, written $\omega(x)$, is either a value (v) or undefined junk (**?**). By $\omega[x \mapsto \cdots]$ (resp. $\omega[x := \cdots]$) I denote the result of extending (resp. updating) the store ω with the binding of x to $\cdots$. The empty store is denoted ϵ.

A machine state Ω is either the error state (Error) or a normal state of the form $(\omega; \mathcal{C}; e)$, where ω is the current store, $\mathcal{C}$ is the current continuation, and e is the term currently being evaluated. A continuation $\mathcal{C}$ consists of a stack of continuation frames $\mathcal{F}$. There are only two forms of continuation frames in the language: one is pushed onto the continuation stack when evaluating a let binding, the other when evaluating the body of a recursive term.

[2] An abstract machine semantics also makes it easier to formalize exception handling in the full IL of Chapter 8.

Machine Stores	ω
Machine States	$\Omega ::= (\omega; \mathcal{C}; e) \mid \mathsf{Error}$
Continuations	$\mathcal{C} ::= \bullet \mid \mathcal{C} \circ \mathcal{F}$
Continuation Frames	$\mathcal{F} ::= \mathsf{let}\ x = \bullet\ \mathsf{in}\ e \mid \mathsf{rec}(x : \mathrm{C}.\, \bullet)$

$\boxed{\textbf{Small-step semantics: } \Omega \mapsto \Omega'}$

$$\frac{}{(\omega; \mathcal{C}; \mathsf{let}\ x = e'\ \mathsf{in}\ e) \mapsto (\omega; \mathcal{C} \circ \mathsf{let}\ x = \bullet\ \mathsf{in}\ e; e')} \qquad \frac{}{(\omega; \mathcal{C} \circ \mathsf{let}\ x = \bullet\ \mathsf{in}\ e; v) \mapsto (\omega; \mathcal{C}; e[v/x])}$$

$$\frac{x \notin \mathrm{dom}(\omega)}{(\omega; \mathcal{C}; \mathsf{rec}(x : \mathrm{C}.\, e)) \mapsto (\omega[x \mapsto \textbf{?}]; \mathcal{C} \circ \mathsf{rec}(x : \mathrm{C}.\, \bullet); e)} \qquad \frac{x \in \mathrm{dom}(\omega)}{(\omega; \mathcal{C} \circ \mathsf{rec}(x : \mathrm{C}.\, \bullet); v) \mapsto (\omega[x := v]; \mathcal{C}; v)}$$

$$\frac{}{(\omega; \mathcal{C}; \mathsf{unfold}_{\mathrm{C}}\langle\!\langle \mathsf{fold}_{\mathrm{D}}\langle\!\langle v \rangle\!\rangle \rangle\!\rangle) \mapsto (\omega; \mathcal{C}; v)}$$

$$\frac{x \in \mathrm{dom}(\omega) \quad \omega(x) = v}{(\omega; \mathcal{C}; \mathsf{fetch}(x)) \mapsto (\omega; \mathcal{C}; v)} \qquad \frac{x \in \mathrm{dom}(\omega) \quad \omega(x) = \textbf{?}}{(\omega; \mathcal{C}; \mathsf{fetch}(x)) \mapsto \mathsf{Error}}$$

For all other rules of the form $\overline{e \mapsto e'}$ from Figure 3.14, we now have:

$$\frac{}{(\omega; \mathcal{C}; e) \mapsto (\omega; \mathcal{C}; e')}$$

Figure 6.9: Abstract Machine Semantics With Explicit Store and Control Stack

$\boxed{\textbf{Well-formed continuations: } \Gamma \vdash C : \mathrm{C}\ \mathsf{cont}}$

$$\frac{}{\Gamma \vdash \bullet : \mathrm{C}\ \mathsf{cont}} \qquad \frac{\Gamma \vdash \mathcal{F} : \mathrm{C} \Rightarrow \mathrm{D} \quad \Gamma \vdash \mathcal{C} : \mathrm{D}\ \mathsf{cont}}{\Gamma \vdash \mathcal{C} \circ \mathcal{F} : \mathrm{C}\ \mathsf{cont}} \qquad \frac{\Gamma \vdash \mathcal{C} : \mathrm{D}\ \mathsf{cont} \quad \Gamma \vdash \mathrm{D} \equiv \mathrm{C} : \mathbf{T}}{\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}}$$

$\boxed{\textbf{Well-formed continuation frames: } \Gamma \vdash \mathcal{F} : \mathrm{C}_1 \Rightarrow \mathrm{C}_2}$

$$\frac{\Gamma, x{:}\mathrm{C}_1 \vdash e : \mathrm{C}_2}{\Gamma \vdash \mathsf{let}\ x = \bullet\ \mathsf{in}\ e : \mathrm{C}_1 \Rightarrow \mathrm{C}_2} \qquad \frac{x{:}\mathsf{maybe}(\mathrm{C}) \in \Gamma}{\Gamma \vdash \mathsf{rec}(x : \mathrm{C}.\, \bullet) : \mathrm{C} \Rightarrow \mathrm{C}}$$

Figure 6.10: Well-Formed Continuations

The definition of well-formed machine state, given below in Definition 6.2.9, relies naturally on a notion of well-formed continuation and well-formed machine store. The former is formalized in Figure 6.10 in a fairly typical way.[3] The latter depends in turn on a notion of "run-time" context, which is a context that only binds variables at types classifying locations in the store. Run-time contexts and well-formed stores are formalized as follows:

Definition 6.2.7 (Run-Time Contexts)
A context Γ is *run-time* if the only bindings in Γ have the form $x{:}\mathsf{maybe}(\mathrm{C})$.

[3] N.B. The only oddity perhaps is the premise $x{:}\mathsf{maybe}(\mathrm{C}) \in \Gamma$ in the recursive term frame rule, which is needed in order to prove progress for the recursive backpatching step. The important point here is that the frame $\mathsf{rec}(x : \mathrm{C}.\, \bullet)$ does *not* bind the variable x. When this frame comes into existence, x refers to a location that has been created in the machine store, and thus it must be bound in the context Γ.

Definition 6.2.8 (Well-Formed Machine Stores)
A machine store ω is *well-formed*, denoted $\Gamma \vdash \omega$, if:

1. Γ is run-time and $\mathrm{dom}(\omega) = \mathrm{dom}(\Gamma)$

2. $\forall x{:}\mathsf{maybe}(\mathrm{C}) \in \Gamma$. either $\omega(x) = \mathbf{?}$ or $\omega(x) = v$, where $\Gamma \vdash v : \mathrm{C}$

We can now define a notion of well-formed machine state, which requires that the type of the term currently being evaluated is the same as the type that the current continuation expects:

Definition 6.2.9 (Well-Formed Machine States)
A machine state Ω is *well-formed*, denoted $\Gamma \vdash \Omega$, if either $\Omega = \mathsf{Error}$ or $\Omega = (\omega; \mathcal{C}; e)$, where:

1. $\Gamma \vdash \omega$

2. $\exists \mathrm{C}.\ \Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}$ and $\Gamma \vdash e : \mathrm{C}$

We can now state the preservation and progress theorems leading to type safety.

Theorem 6.2.10 (Preservation)
If $\Gamma \vdash \Omega$ and $\Omega \mapsto \Omega'$, then $\exists\, \Gamma'.\ \Gamma' \vdash \Omega'$.

Definition 6.2.11 (Terminal States)
A machine state Ω is *terminal* if it has the form Error or $(\omega; \bullet; v)$.

Definition 6.2.12 (Stuck States)
A machine state Ω is *stuck* if it is non-terminal and there is no state Ω' such that $\Omega \mapsto \Omega'$.

Theorem 6.2.13 (Progress)
If $\Gamma \vdash \Omega$, then Ω is not stuck.

Corollary 6.2.14 (Type Safety)
If $\emptyset \vdash e : \mathrm{C}$, then the evaluation of $(\epsilon; \bullet; e)$ never enters a stuck state.

Lastly, the proof of progress relies on the following extension of the canonical forms lemma from Section 3.2.5:

Lemma 6.2.15 (Canonical Forms)
Suppose that Γ is run-time and $\Gamma \vdash v \overset{\mathrm{wh}}{\Longrightarrow} \mathrm{C}$.

1. If C is of the form $\mathsf{maybe}(\mathrm{D})$, then v is of the form x.

2. If C is of the form $\mathrm{C}_1 \rightsquigarrow \mathrm{C}_2$, then v is of the form $\mathsf{fold}_{\mathrm{D}}$ or $\mathsf{unfold}_{\mathrm{D}}$.

3. If C is of the form Q, then v is of the form $\mathsf{fold}_{\mathrm{D}}\langle\!\langle v' \rangle\!\rangle$.

4. All other cases are the same as in Lemma 3.2.8.

$$\begin{array}{lrcl} \text{Signatures} & \mathrm{S}, \mathrm{R} & ::= & \cdots \mid \mathsf{maybe}(\mathrm{S}) \mid \rho\,\mathrm{X}.\mathrm{S} \\ \text{Transparent Signatures} & \mathbb{S}, \mathbb{R} & ::= & \cdots \mid \mathsf{maybe}(\mathbb{S}) \mid \rho\,\mathrm{X}.\mathbb{S} \end{array}$$

$$\begin{array}{lcl} \mathsf{Fst}(\mathsf{maybe}(\mathrm{S})) & \stackrel{\text{def}}{=} & \mathsf{Fst}(\mathrm{S}) \\ \mathsf{Fst}(\rho\,\mathrm{X}.\mathrm{S}) & \stackrel{\text{def}}{=} & \mathsf{Fst}(\mathrm{S}),\ \text{assuming } \mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\mathsf{Fst}(\mathrm{S})) \end{array}$$

$$\begin{array}{lcl} \mathfrak{s}_{\mathsf{maybe}(\mathrm{S})}(\mathrm{C}) & \stackrel{\text{def}}{=} & \mathsf{maybe}(\mathfrak{s}_{\mathrm{S}}(\mathrm{C})) \\ \mathfrak{s}_{\rho\,\mathrm{X}.\mathrm{S}}(\mathrm{C}) & \stackrel{\text{def}}{=} & \rho\,(\mathfrak{s}_{\mathrm{S}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C})) \end{array}$$

Figure 6.11: Extensions to Signature Syntax and Related Functions

6.3 Signature-Language Extensions

Figure 6.11 shows the extensions to the language of signatures from Chapter 4. There are two new signature forms: the $\mathsf{maybe}(\mathrm{S})$ signature used to classify recursive module variables, and the recursively dependent signature $\rho\,\mathrm{X}.\mathrm{S}$. I will write $\rho\,(\mathrm{S})$ as shorthand for $\rho\,\mathrm{X}.\mathrm{S}$ when $\mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\mathrm{S})$, *i.e.,* when the rds is "degenerate."

Figure 6.11 also gives the corresponding extensions to the definitions of $\mathsf{Fst}(\mathrm{S})$ and $\mathfrak{s}_{\mathrm{S}}(\mathrm{C})$. Let us begin with $\mathsf{maybe}(\mathrm{S})$. Extensionally speaking, a module M of signature $\mathsf{maybe}(\mathrm{S})$ is like a functor of signature $1 \xrightarrow{\mathsf{tot}} \mathrm{S}$. There is only one thing we can do with it—fetch it, which may raise a run-time error—just as there is only one thing we can do with a functor with unit argument, and that is to apply it to $\langle\rangle$. The analogy is to a total, rather than a partial, functor, because every $\mathsf{fetch}(\mathrm{M})$ returns the same module value with the same type components, if it returns at all. This analogy would suggest that $\mathsf{Fst}(\mathsf{maybe}(\mathrm{S}))$ be defined as $1 \rightarrow \mathsf{Fst}(\mathrm{S})$, and $\mathfrak{s}_{\mathsf{maybe}(\mathrm{S})}(\mathrm{C})$ as $\mathsf{maybe}(\mathfrak{s}_{\mathrm{S}}(\mathrm{C}\langle\rangle))$. The definition in Figure 6.11 takes the extra step of reducing $\mathsf{Fst}(\mathsf{maybe}(\mathrm{S}))$ from $1 \rightarrow \mathsf{Fst}(\mathrm{S})$ to $\mathsf{Fst}(\mathrm{S})$, as the two kinds are isomorphic.

The rds construct $\rho\,\mathrm{X}.\mathrm{S}$ describes modules $\mathbb{M}$ of signature $\mathrm{S}[\mathbb{M}/\mathrm{X}]$. The dynamic-on-static restriction (discussed extensively in Chapter 5) ensures that $\mathsf{Fst}(\mathrm{S})$ may not refer to X^{c} and therefore that $\mathsf{Fst}(\mathrm{S}[\mathbb{M}/\mathrm{X}]) = \mathsf{Fst}(\mathrm{S})$. Correspondingly, $\mathsf{Fst}(\rho\,\mathrm{X}.\mathrm{S})$ is defined simply as $\mathsf{Fst}(\mathrm{S})$. As for the singleton signature $\mathfrak{s}_{\rho\mathrm{X}.\mathrm{S}}(\mathrm{C})$, it is intended to describe modules $\mathbb{M}$ of signature $\mathrm{S}[\mathbb{M}/\mathrm{X}]$ whose static part $\mathsf{Fst}(\mathbb{M})$ is equivalent to C—that is, modules $\mathbb{M}$ of signature $\mathfrak{s}_{\mathrm{S}[\mathbb{M}/\mathrm{X}]}(\mathrm{C})$. This latter signature is in turn equivalent to $\mathfrak{s}_{\mathrm{S}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C})$, since $\mathsf{Fst}(\mathbb{M})$ is equivalent to C.

Note that the definition of $\mathfrak{s}_{\rho\,\mathrm{X}.\mathrm{S}}(\mathrm{C})$ in Figure 6.11 places $\mathfrak{s}_{\mathrm{S}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C})$ inside a degenerate rds $\rho\,(\cdot)$. Morally, for any S, the signatures S and $\rho\,(\mathrm{S})$ are equivalent. In this calculus, however, I have chosen to distinguish them and introduce explicit introduction and elimination forms for rds's (see Section 6.4 below).[4] This approach allows us to maintain the syntax-directed nature of signature equivalence/subtyping, which simplifies the meta-theory. Since rds's are only considered subtypes of other rds's, the definition of $\mathfrak{s}_{\rho\,\mathrm{X}.\mathrm{S}}(\mathrm{C})$ is wrapped in a degenerate rds in order to preserve the property that $\mathfrak{s}_{\mathrm{S}}(\mathrm{C})$ is a subtype of S.

Figure 6.12 gives the well-formedness, equivalence and subtyping rules for the new signature constructs. The rules for $\mathsf{maybe}(\mathrm{S})$ signatures are all obvious. As for rds's, the well-formedness rule (Rule 117) is precisely the one given in Section 5.2.2 of the previous chapter. The rds subtyping and equivalence rules, though, are rather unusual: what are singleton signatures doing in

[4]The programmer of the external language defined in Chapter 9 will not have to write these explicit coercions herself—the elaborator infers them as part of signature matching.

Well-formed signatures: $\Delta \vdash \mathrm{S}\ \mathrm{sig}$

$$\frac{\Delta \vdash \mathrm{S}\ \mathrm{sig}}{\Delta \vdash \mathsf{maybe}(\mathrm{S})\ \mathrm{sig}}\ (116) \qquad \frac{\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}) \vdash \mathrm{S}\ \mathrm{sig}}{\Delta \vdash \rho\,\mathrm{X.S}\ \mathrm{sig}}\ (117)$$

Signature equivalence: $\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$

$$\frac{\Delta \vdash \mathrm{S}_1 \equiv \mathrm{S}_2}{\Delta \vdash \mathsf{maybe}(\mathrm{S}_1) \equiv \mathsf{maybe}(\mathrm{S}_2)}\ (118)$$

$$\frac{\begin{array}{c}\Delta \vdash \rho\,\mathrm{X.S}_1\ \mathrm{sig} \quad \Delta \vdash \rho\,\mathrm{X.S}_2\ \mathrm{sig}\\ \Delta \vdash \mathsf{Fst}(\mathrm{S}_1) \equiv \mathsf{Fst}(\mathrm{S}_2) \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1) \vdash \mathfrak{s}_{\mathrm{S}_1}(\mathrm{X}^{\mathsf{c}}) \equiv \mathfrak{s}_{\mathrm{S}_2}(\mathrm{X}^{\mathsf{c}})\end{array}}{\Delta \vdash \rho\,\mathrm{X.S}_1 \equiv \rho\,\mathrm{X.S}_2}\ (119)$$

Signature subtyping: $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$

$$\frac{\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2}{\Delta \vdash \mathsf{maybe}(\mathrm{S}_1) \leq \mathsf{maybe}(\mathrm{S}_2)}\ (120)$$

$$\frac{\begin{array}{c}\Delta \vdash \rho\,\mathrm{X.S}_1\ \mathrm{sig} \quad \Delta \vdash \rho\,\mathrm{X.S}_2\ \mathrm{sig}\\ \Delta \vdash \mathsf{Fst}(\mathrm{S}_1) \leq \mathsf{Fst}(\mathrm{S}_2) \quad \Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1) \vdash \mathfrak{s}_{\mathrm{S}_1}(\mathrm{X}^{\mathsf{c}}) \leq \mathfrak{s}_{\mathrm{S}_2}(\mathrm{X}^{\mathsf{c}})\end{array}}{\Delta \vdash \rho\,\mathrm{X.S}_1 \leq \rho\,\mathrm{X.S}_2}\ (121)$$

Figure 6.12: New Inference Rules for Signatures

the last premise? Wouldn't a more natural premise (in the rds subtyping rule, for instance) be $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1) \vdash \mathrm{S}_1 \leq \mathrm{S}_2$?

The problem with this simpler, more obvious premise is that it is considerably more restrictive than necessary. For example, consider the signatures $\mathrm{R}_1 = \rho\,\mathrm{X.S}_1$ and $\mathrm{R}_2 = \rho\,\mathrm{X.S}_2$, where

$$\begin{array}{lcl}\mathrm{S}_1 & \stackrel{\mathrm{def}}{=} & \Sigma\mathrm{Y}{:}[\![\mathbf{T}]\!].[\![\pi_1\mathrm{X}^{\mathsf{c}} \times \mathrm{Y}^{\mathsf{c}}]\!]\\ \mathrm{S}_2 & \stackrel{\mathrm{def}}{=} & \Sigma\mathrm{Y}{:}[\![\mathbf{T}]\!].[\![\mathrm{Y}^{\mathsf{c}} \times \pi_1\mathrm{X}^{\mathsf{c}}]\!]\end{array}$$

Written in pseudo-ML syntax, these would correspond to

$$\begin{array}{lcl}\mathrm{R}_1 & \stackrel{\mathrm{def}}{=} & \rho\texttt{X. sig type t ; val x : X.t * t end}\\ \mathrm{R}_2 & \stackrel{\mathrm{def}}{=} & \rho\texttt{X. sig type t ; val x : t * X.t end}\end{array}$$

Intuitively, when checking whether R_1 is a subsignature of R_2, `t` and `X.t` should be treated as equivalent because they are really different ways of referring to the same type component. However, the premise $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1) \vdash \mathrm{S}_1 \leq \mathrm{S}_2$ does not identify them; it simply compares S_1 and S_2 directly, and in the above example S_1 and S_2 are incomparable.

Rules 119 and 121 address this limitation by comparing $\mathfrak{s}_{\mathrm{S}_1}(\mathrm{X}^{\mathsf{c}})$ and $\mathfrak{s}_{\mathrm{S}_2}(\mathrm{X}^{\mathsf{c}})$ instead of S_1 and S_2. In terms of the example above, this approach eliminates the distinction between `t` and `X.t`. Formally, R_1 and R_2 will be deemed equivalent since $\mathfrak{s}_{\mathrm{S}_1}(\mathrm{X}^{\mathsf{c}}) = \mathfrak{s}_{\mathrm{S}_2}(\mathrm{X}^{\mathsf{c}}) = [\![\mathfrak{s}(\pi_1\mathrm{X}^{\mathsf{c}})]\!] \times [\![\pi_1\mathrm{X}^{\mathsf{c}} \times \pi_1\mathrm{X}^{\mathsf{c}}]\!]$.

The new signature forms introduced in this section do not cause any meta-theoretic problems. Nonetheless, since the subtyping and equivalence rules for rds's are so unusual, I will give here the proofs for the rds cases in several of the declarative properties from Section 4.1.3. I will omit the proofs for the $\mathsf{maybe}(\mathrm{S})$ cases, which are all straightforward.

Definition 4.1.2 (Sizes of Signatures)

$$\begin{array}{lcl} \mathrm{size}(\mathsf{maybe}(\mathrm{S})) & \stackrel{\mathrm{def}}{=} & 1 + \mathrm{size}(\mathrm{S}) \\ \mathrm{size}(\rho\,\mathrm{X.S}) & \stackrel{\mathrm{def}}{=} & 1 + \mathrm{size}(\mathrm{S}) \end{array}$$

Proposition 4.1.3 (Facts About $\mathsf{Fst}(\mathrm{S})$ and $\mathfrak{s}_{\mathrm{S}}(\mathrm{C})$)

1. If $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathrm{S})$, then $\Delta \vdash \mathsf{Fst}(\mathfrak{s}_{\mathrm{S}}(\mathrm{C})) \equiv \mathfrak{s}_{\mathsf{Fst}(\mathrm{S})}(\mathrm{C})$.

3. If $\Delta \vdash \mathrm{S}$ sig and $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathrm{S})$, then $\Delta \vdash \mathfrak{s}_{\mathrm{S}}(\mathrm{C})$ sig.

Proof: New case: $\mathrm{S} = \rho\,\mathrm{X.R}$, so $\mathsf{Fst}(\mathrm{S}) = \mathsf{Fst}(\mathrm{R})$ and $\mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\mathsf{Fst}(\mathrm{R}))$.

1. (a) By definition, $\mathsf{Fst}(\mathfrak{s}_{\mathrm{S}}(\mathrm{C})) = \mathsf{Fst}(\rho\,(\mathfrak{s}_{\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C}))) = \mathsf{Fst}(\mathfrak{s}_{\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C}))$.
   (b) By induction, $\Delta \vdash \mathsf{Fst}(\mathfrak{s}_{\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C})) \equiv \mathfrak{s}_{\mathsf{Fst}(\mathrm{R})[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C})$.
   (c) Since $\mathrm{X}^{\mathsf{c}} \notin \mathrm{FV}(\mathsf{Fst}(\mathrm{R}))$, we have $\mathfrak{s}_{\mathsf{Fst}(\mathrm{R})[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C}) = \mathfrak{s}_{\mathsf{Fst}(\mathrm{S})}(\mathrm{C})$.

3. (a) By definition, we need to show that $\Delta \vdash \rho\,(\mathfrak{s}_{\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C}))$ sig.
   (b) First, $\mathsf{Fst}(\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]) = \mathsf{Fst}(\mathrm{R})$, so by assumption, $\Delta \vdash \mathsf{Fst}(\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}])$ kind.
   (c) Second, since $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{R}) \vdash \mathrm{R}$ sig and $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathrm{R})$,
   (d) By Substitution, $\Delta \vdash \mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]$ sig.
   (e) By induction, $\Delta \vdash \mathfrak{s}_{\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C})$ sig, so $\Delta \vdash \rho\,(\mathfrak{s}_{\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C}))$ sig.

■

Proposition 4.1.4 (Reflexivity)
If $\Delta \vdash \mathrm{S}$ sig, then $\Delta \vdash \mathrm{S} \equiv \mathrm{S}$ and $\Delta \vdash \mathrm{S} \leq \mathrm{S}$.

Proof: By induction on $\mathrm{size}(\mathrm{S})$. New case: $\mathrm{S} = \rho\,\mathrm{X.R}$.

1. By assumption, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{R}) \vdash \mathrm{R}$ sig.
2. By Reflexivity, $\Delta \vdash \mathsf{Fst}(\mathrm{R}) \equiv \mathsf{Fst}(\mathrm{R})$.
3. By Part 3 of Proposition 4.1.3, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{R}) \vdash \mathfrak{s}_{\mathrm{R}}(\mathrm{X}^{\mathsf{c}})$ sig.
4. Since $\mathrm{size}(\mathfrak{s}_{\mathrm{R}}(\mathrm{X}^{\mathsf{c}})) = \mathrm{size}(\mathrm{R}) < \mathrm{size}(\mathrm{S})$, by induction, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{R}) \vdash \mathfrak{s}_{\mathrm{R}}(\mathrm{X}^{\mathsf{c}}) \equiv \mathfrak{s}_{\mathrm{R}}(\mathrm{X}^{\mathsf{c}})$.
5. Thus, $\Delta \vdash \rho\,\mathrm{X.R} \equiv \rho\,\mathrm{X.R}$.

The proof of $\Delta \vdash \rho\,\mathrm{X.R} \leq \rho\,\mathrm{X.R}$ is similar. ■

Proposition 4.1.9 (Singleton and Transparent Signature Rules)

1. If $\Delta \vdash \mathrm{S}$ sig and $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathrm{S})$, then $\Delta \vdash \mathfrak{s}_{\mathrm{S}}(\mathrm{C}) \leq \mathrm{S}$.

2. If $\Delta \vdash \mathbb{S}$ sig and $\Delta \vdash \mathrm{C} : \mathsf{Fst}(\mathbb{S})$, then $\Delta \vdash \mathfrak{s}_{\mathbb{S}}(\mathrm{C}) \equiv \mathbb{S}$.

3. If $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$ and $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathsf{Fst}(\mathrm{S}_1)$, then $\Delta \vdash \mathfrak{s}_{\mathrm{S}_1}(\mathrm{C}_1) \leq \mathfrak{s}_{\mathrm{S}_2}(\mathrm{C}_2)$.

$\boxed{\textbf{Signature phase-splitting: } \mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}]\!]}$

$$\frac{\mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}]\!]}{\mathsf{maybe}(\mathrm{S}) \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathsf{maybe}(\mathrm{C})]\!]} \qquad \frac{\mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}]\!]}{\rho\,\mathrm{X}.\mathrm{S} \rightrightarrows [\![\alpha{:}\mathrm{K}.\mathrm{C}[\alpha/\mathrm{X}^{\mathsf{c}}]]\!]}$$

Figure 6.13: New Signature Phase-Splitting Rules

Proof: New case: $\mathrm{S} = \rho\,\mathrm{X}.\mathrm{R}$, $\mathrm{S}_i = \rho\,\mathrm{X}.\mathrm{R}_i$. The proof makes use inductively of Corollary 6.3.1, shown below.

1. (a) Let $\mathbb{R}' = \mathfrak{s}_{\mathrm{R}[\mathrm{C}/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C})$.
   (b) Since $\mathfrak{s}_{\mathrm{S}}(\mathrm{C}) = \rho\,(\mathbb{R}')$, we need to show that $\Delta \vdash \rho\,(\mathbb{R}') \leq \rho\,\mathrm{X}.\mathrm{R}$.
   (c) First, by Proposition 4.1.3, $\Delta \vdash \mathsf{Fst}(\mathbb{R}') \equiv \mathfrak{s}_{\mathsf{Fst}(\mathrm{R})}(\mathrm{C})$.
   (d) By Proposition 3.1.12, $\Delta \vdash \mathfrak{s}_{\mathsf{Fst}(\mathrm{R})}(\mathrm{C}) \leq \mathsf{Fst}(\mathrm{R})$.
   (e) Thus, by Transitivity, $\Delta \vdash \mathsf{Fst}(\mathbb{R}') \leq \mathsf{Fst}(\mathrm{R})$.
   (f) Second, by Proposition 4.1.3 and Reflexivity, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}') \vdash \mathfrak{s}_{\mathrm{R}}(\mathrm{X}^{\mathsf{c}}) \equiv \mathfrak{s}_{\mathrm{R}}(\mathrm{X}^{\mathsf{c}})$.
   (g) Since $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}') \vdash \mathrm{C} \equiv \mathrm{X}^{\mathsf{c}} : \mathsf{Fst}(\mathbb{R}')$, by Functionality, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}') \vdash \mathbb{R}' \equiv \mathfrak{s}_{\mathrm{R}}(\mathrm{X}^{\mathsf{c}})$.
   (h) By induction, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}') \vdash \mathfrak{s}_{\mathrm{R}}(\mathrm{X}^{\mathsf{c}}) \leq \mathrm{R}$.
   (i) Thus, by Transitivity, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}') \vdash \mathbb{R}' \leq \mathrm{R}$.
   (j) Inductively, by Corollary 6.3.1, $\Delta \vdash \rho\,(\mathbb{R}') \leq \rho\,\mathrm{X}.\mathrm{R}$.

2. Analogous to the proof of Part 1, replacing occurrences of $\leq$ with $\equiv$.

3. (a) Let $\mathbb{R}'_i = \mathfrak{s}_{\mathrm{R}_i[\mathrm{C}_i/\mathrm{X}^{\mathsf{c}}]}(\mathrm{C}_i)$.
   (b) Since $\mathfrak{s}_{\mathrm{S}_i}(\mathrm{C}_i) = \rho\,(\mathbb{R}'_i)$, we need to show that $\Delta \vdash \rho\,(\mathbb{R}'_1) \leq \rho\,(\mathbb{R}'_2)$.
   (c) By assumption, $\Delta \vdash \mathsf{Fst}(\mathrm{R}_1) \leq \mathsf{Fst}(\mathrm{R}_2)$ and $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{R}_1) \vdash \mathfrak{s}_{\mathrm{R}_1}(\mathrm{X}^{\mathsf{c}}) \leq \mathfrak{s}_{\mathrm{R}_2}(\mathrm{X}^{\mathsf{c}})$.
   (d) Since $\Delta \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathsf{Fst}(\mathrm{R}_1)$, by Functionality, $\Delta \vdash \mathbb{R}'_1 \leq \mathbb{R}'_2$.
   (e) Inductively, by Corollary 6.3.1, $\Delta \vdash \rho\,(\mathbb{R}'_1) \leq \rho\,(\mathbb{R}'_2)$.

■

Corollary 6.3.1 (Admissible Rds Rules)
Suppose $\Delta \vdash \rho\,\mathrm{X}.\mathrm{S}_1$ sig and $\Delta \vdash \rho\,\mathrm{X}.\mathrm{S}_2$ sig.

1. If $\Delta \vdash \mathsf{Fst}(\mathrm{S}_1) \equiv \mathsf{Fst}(\mathrm{S}_2)$ and $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1) \vdash \mathrm{S}_1 \equiv \mathrm{S}_2$, then $\Delta \vdash \rho\,\mathrm{X}.\mathrm{S}_1 \equiv \rho\,\mathrm{X}.\mathrm{S}_2$.
2. If $\Delta \vdash \mathsf{Fst}(\mathrm{S}_1) \leq \mathsf{Fst}(\mathrm{S}_2)$ and $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1) \vdash \mathrm{S}_1 \leq \mathrm{S}_2$, then $\Delta \vdash \rho\,\mathrm{X}.\mathrm{S}_1 \leq \rho\,\mathrm{X}.\mathrm{S}_2$.

Proof: Follows directly from Part 3 of Proposition 4.1.9. ■

Figure 6.13 presents the phase-splitting rules for the new signature constructs. The rule for $\mathsf{maybe}(\mathrm{S})$ makes clear that the maybe only applies to the dynamic part of S. The rule for $\rho\,\mathrm{X}.\mathrm{S}$ is the same as the one given in Section 5.2.2; thanks to the dynamic-on-static restriction, all recursive references to X in S appear in the dynamic part of S, so the phase-splitting rule can replace them

Modules	$\mathrm{M}, \mathrm{N}, \mathrm{F} ::= \cdots \mid \mathsf{rec}(\mathrm{X} : \mathbb{S}.\, \mathrm{M}) \mid \mathsf{fetch}(\mathrm{M}) \mid \mathsf{roll}(\mathrm{M}) \mid \mathsf{unroll}(\mathrm{M})$
Projectible Modules	$\mathbb{M}, \mathbb{N}, \mathbb{F} ::= \cdots \mid \mathsf{rec}(\mathrm{X} : \mathbb{S}.\, \mathrm{M}) \mid \mathsf{fetch}(\mathbb{M}) \mid \mathsf{roll}(\mathbb{M}) \mid \mathsf{unroll}(\mathbb{M})$

$$
\begin{array}{lcl}
\mathsf{Fst}(\mathsf{rec}(\mathrm{X} : \mathbb{S}.\, \mathrm{M})) & \stackrel{\text{def}}{=} & \mathsf{Can}(\mathsf{Fst}(\mathbb{S})) \\
\mathsf{Fst}(\mathsf{fetch}(\mathbb{M})) & \stackrel{\text{def}}{=} & \mathsf{Fst}(\mathbb{M}) \\
\mathsf{Fst}(\mathsf{roll}(\mathbb{M})) & \stackrel{\text{def}}{=} & \mathsf{Fst}(\mathbb{M}) \\
\mathsf{Fst}(\mathsf{unroll}(\mathbb{M})) & \stackrel{\text{def}}{=} & \mathsf{Fst}(\mathbb{M})
\end{array}
$$

Figure 6.14: Extensions to Module Syntax

with direct references to the variable α. I give here the proof of the one new and interesting case in the soundness theorem.

Proposition 4.1.10 (Soundness and Other Properties of Signature Phase-Splitting)

5. If $\Delta \vdash \mathrm{S}_1 \leq \mathrm{S}_2$ and $\mathrm{S}_1 \rightrightarrows [\![\alpha{:}\mathrm{K}_1.\mathrm{C}_1]\!]$ and $\mathrm{S}_2 \rightrightarrows [\![\alpha{:}\mathrm{K}_2.\mathrm{C}_2]\!]$, then $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2$ and $\Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}$.

Proof:

5. New case: $\mathrm{S}_i = \rho\, \mathrm{X}.\mathrm{R}_i$ and $\mathrm{S}_i \rightrightarrows [\![\alpha{:}\mathrm{K}_i.\mathrm{C}_i[\alpha/\mathrm{X}^{\mathsf{c}}]]\!]$, where $\mathrm{R}_i \rightrightarrows [\![\alpha{:}\mathrm{K}_i.\mathrm{C}_i]\!]$.
   (a) By assumption, $\Delta \vdash \mathsf{Fst}(\mathrm{R}_1) \leq \mathsf{Fst}(\mathrm{R}_2)$ and $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{R}_1) \vdash \mathfrak{s}_{\mathrm{R}_1}(\mathrm{X}^{\mathsf{c}}) \leq \mathfrak{s}_{\mathrm{R}_2}(\mathrm{X}^{\mathsf{c}})$.
   (b) By Part 2 of this proposition, $\mathrm{K}_i = \mathsf{Fst}(\mathrm{R}_i)$, so $\Delta \vdash \mathrm{K}_1 \leq \mathrm{K}_2$.
   (c) By induction, $\mathfrak{s}_{\mathrm{R}_i}(\mathrm{X}^{\mathsf{c}}) \rightrightarrows [\![\alpha{:}\mathbb{L}_i.\mathrm{D}_i]\!]$,
   (d) where $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1 \vdash \mathbb{L}_1 \leq \mathbb{L}_2$ and $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1, \alpha{:}\mathbb{L}_1 \vdash \mathrm{D}_1 \equiv \mathrm{D}_2 : \mathbf{T}$.
   (e) By Proposition 4.1.9, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1 \vdash \mathfrak{s}_{\mathrm{R}_i}(\mathrm{X}^{\mathsf{c}}) \leq \mathrm{R}_i$.
   (f) By induction, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1 \vdash \mathbb{L}_i \leq \mathrm{K}_i$ and $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1, \alpha{:}\mathbb{L}_i \vdash \mathrm{D}_i \equiv \mathrm{C}_i : \mathbf{T}$.
   (g) By Transitivity, $\Delta, \mathrm{X}^{\mathsf{c}}{:}\mathrm{K}_1, \alpha{:}\mathbb{L}_1 \vdash \mathrm{C}_1 \equiv \mathrm{C}_2 : \mathbf{T}$.
   (h) By Substitution, $\Delta, \alpha{:}\mathrm{K}_1 \vdash \mathrm{C}_1[\alpha/\mathrm{X}^{\mathsf{c}}] \equiv \mathrm{C}_2[\alpha/\mathrm{X}^{\mathsf{c}}] : \mathbf{T}$.

■

6.4 Module-Language Extensions

Figure 6.14 shows the extensions to the syntax of modules. The recursive module construct $\mathsf{rec}(\mathrm{X} : \mathbb{S}.\, \mathrm{M})$ requires the declared signature to be transparent. As explained in Chapter 5, this is the only simple, type-theoretic approach I know of for addressing the double vision problem. Since $\mathsf{rec}(\mathrm{X} : \mathbb{S}.\, \mathrm{M})$ is transparent, it is considered projectible as well. The static part of $\mathsf{rec}(\mathrm{X} : \mathbb{S}.\, \mathrm{M})$ is defined as the canonical implementation of $\mathsf{Fst}(\mathbb{S})$, although any implementation of $\mathsf{Fst}(\mathbb{S})$ will do.

The module $\mathsf{fetch}(\mathrm{M})$ evaluates M to a memory location and then dereferences the memory location. If the contents of the location are undefined, then a run-time error is raised. As discussed in Section 6.3, $\mathsf{Fst}(\mathsf{maybe}(\mathrm{S})) = \mathsf{Fst}(\mathrm{S})$. Correspondingly, $\mathsf{fetch}(\mathrm{M})$ is projectible so long as M is, and $\mathsf{Fst}(\mathsf{fetch}(\mathrm{M})) = \mathsf{Fst}(\mathrm{M})$.

Well-formed modules: $\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}$

$$\frac{\Gamma, \mathrm{X}{:}\mathsf{maybe}(\mathbb{S}) \vdash \mathrm{M} :_\mathsf{P} \mathbb{S}}{\Gamma \vdash \mathsf{rec}(\mathrm{X}:\mathbb{S}.\,\mathrm{M}) :_\mathsf{P} \mathbb{S}} \;(122) \qquad \frac{\Gamma \vdash \mathrm{M} :_\kappa \mathsf{maybe}(\mathrm{S})}{\Gamma \vdash \mathsf{fetch}(\mathrm{M}) :_\kappa \mathrm{S}} \;(123)$$

$$\frac{\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}}{\Gamma \vdash \mathsf{roll}(\mathrm{M}) :_\kappa \rho\,(\mathrm{S})} \;(124) \qquad \frac{\Gamma \vdash \mathbb{M} :_\mathsf{P} \rho\,\mathrm{X}.\mathrm{S}}{\Gamma \vdash \mathsf{unroll}(\mathbb{M}) :_\mathsf{P} \mathrm{S}[\mathbb{M}/\mathrm{X}]} \;(125)$$

Figure 6.15: New Inference Rules for Modules

The modules $\mathsf{roll}(\mathrm{M})$ and $\mathsf{unroll}(\mathrm{M})$ are the introduction and elimination forms for rds's, respectively. They are merely signature coercions and have no run-time effect. Thus, as the phase-splitting rules for modules will evidence, $\mathsf{roll}(\mathrm{M})$ and $\mathsf{unroll}(\mathrm{M})$ have exactly the same core-language interpretation as M does, and $\mathsf{Fst}(\mathsf{roll}(\mathbb{M})) = \mathsf{Fst}(\mathsf{unroll}(\mathbb{M})) = \mathsf{Fst}(\mathbb{M})$.

Figure 6.15 gives the typing rules for these new module constructs. Rule 122 for $\mathsf{rec}(\mathrm{X}:\mathbb{S}.\,\mathrm{M})$ requires that M have signature $\mathbb{S}$, assuming X is a location that will eventually store the result of evaluating M. Rule 123 says that $\mathsf{fetch}(\mathrm{M})$ is pure if M is. Rule 124 for $\mathsf{roll}(\mathrm{M})$ coerces M into an rds by wrapping the signature of M in a degenerate rds. Rule 125 restricts the argument of $\mathsf{unroll}(\mathbb{M})$ to be projectible, so that $\mathsf{Fst}(\mathbb{M})$ may be substituted for X^c in the body of the rds $\rho\,\mathrm{X}.\mathrm{S}$ classifying $\mathbb{M}$.

It is easy to show that the declarative properties of modules given in Section 4.2.4 are preserved by the present extensions. One point of note: the rule for rds introduction given here as Rule 124 is simpler than the rule suggested initially in Section 5.2.2. That earlier rule, which was essentially the inversion of the unroll rule, is admissible in the present system:

Proposition 6.4.1 (Admissible Roll Rule)
If $\Gamma \vdash \rho\,\mathrm{X}.\mathrm{S}$ sig and $\Gamma \vdash \mathbb{M} :_\mathsf{P} \mathrm{S}[\mathbb{M}/\mathrm{X}]$, then $\Gamma \vdash \mathsf{roll}(\mathbb{M}) :_\mathsf{P} \rho\,\mathrm{X}.\mathrm{S}$.

Proof:

1. Let $\mathbb{R} = \mathfrak{s}_{\mathrm{S}[\mathsf{Fst}(\mathbb{M})/\mathrm{X}^\mathsf{c}]}(\mathsf{Fst}(\mathbb{M}))$.
2. Since $\Gamma \vdash \mathbb{M} :_\mathsf{P} \mathrm{S}[\mathbb{M}/\mathrm{X}]$, by Rule 102, $\Gamma \vdash \mathbb{M} :_\mathsf{P} \mathbb{R}$.
3. Thus, by Rule 124, $\Gamma \vdash \mathsf{roll}(\mathbb{M}) :_\mathsf{P} \rho\,(\mathbb{R})$. We need to show that $\Gamma \vdash \rho\,(\mathbb{R}) \leq \rho\,\mathrm{X}.\mathrm{S}$.
4. First, since $\Gamma \vdash \rho\,\mathrm{X}.\mathrm{S}$ sig, we have $\Gamma \vdash \mathsf{Fst}(\mathrm{S})$ kind.
5. Thus, $\mathrm{X}^\mathsf{c} \notin \mathrm{FV}(\mathsf{Fst}(\mathrm{S}))$, and $\mathsf{Fst}(\mathrm{S}[\mathbb{M}/\mathrm{X}]) = \mathsf{Fst}(\mathrm{S})[\mathbb{M}/\mathrm{X}] = \mathsf{Fst}(\mathrm{S})$.
6. By Proposition 4.1.3, $\Gamma \vdash \mathsf{Fst}(\mathbb{R}) \equiv \mathfrak{s}_{\mathsf{Fst}(\mathrm{S})}(\mathsf{Fst}(\mathbb{M}))$.
7. By Proposition 4.2.3, $\Gamma \vdash \mathsf{Fst}(\mathbb{M}) : \mathsf{Fst}(\mathrm{S})$.
8. Thus, by Proposition 3.1.12, $\Gamma \vdash \mathsf{Fst}(\mathbb{M}) : \mathsf{Fst}(\mathbb{R})$ and $\Gamma \vdash \mathsf{Fst}(\mathbb{R}) \leq \mathsf{Fst}(\mathrm{S})$.
9. By Proposition 3.1.14, $\Gamma, \mathrm{X}^\mathsf{c}{:}\mathsf{Fst}(\mathbb{R}) \vdash \mathsf{Fst}(\mathbb{M}) \equiv \mathrm{X}^\mathsf{c} : \mathsf{Fst}(\mathbb{R})$.
10. Since $\Gamma, \mathrm{X}^\mathsf{c}{:}\mathsf{Fst}(\mathbb{R}) \vdash \mathfrak{s}_\mathrm{S}(\mathrm{X}^\mathsf{c}) \leq \mathrm{S}$, by Functionality, $\Gamma, \mathrm{X}^\mathsf{c}{:}\mathsf{Fst}(\mathbb{R}) \vdash \mathbb{R} \leq \mathrm{S}$.
11. Thus, by Corollary 6.3.1, $\Gamma \vdash \rho\,(\mathbb{R}) \leq \rho\,\mathrm{X}.\mathrm{S}$.

■

Signature synthesis: $\Gamma \vdash \mathrm{M} \Rightarrow_\kappa \mathrm{S}$

$$\frac{\Gamma \vdash \mathbb{S}\ \mathrm{sig} \quad \Gamma, \mathrm{X}{:}\mathsf{maybe}(\mathbb{S}) \vdash \mathrm{M} \Leftarrow_{\mathsf{P}} \mathbb{S}}{\Gamma \vdash \mathsf{rec}(\mathrm{X}:\mathbb{S}.\,\mathrm{M}) \Rightarrow_{\mathsf{P}} \mathbb{S}} \qquad \frac{\Gamma \vdash \mathrm{M} \Rightarrow_\kappa \mathsf{maybe}(\mathrm{S})}{\Gamma \vdash \mathsf{fetch}(\mathrm{M}) \Rightarrow_\kappa \mathrm{S}}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_\kappa \mathrm{S}}{\Gamma \vdash \mathsf{roll}(\mathrm{M}) \Rightarrow_\kappa \rho\,(\mathrm{S})} \qquad \frac{\Gamma \vdash \mathbb{M} \Rightarrow_{\mathsf{P}} \rho\,\mathrm{X}.\mathrm{S}}{\Gamma \vdash \mathsf{unroll}(\mathbb{M}) \Rightarrow_{\mathsf{P}} \mathrm{S}[\mathbb{M}/\mathrm{X}]}$$

Figure 6.16: Extensions to Signature Synthesis

Figure 6.16 shows how to extend the signature synthesis algorithm to handle the new module constructs. All the new synthesis rules are straightforward. Here are the new cases in the proof that the signature checking algorithm is complete:

Theorem 4.2.6 (Completeness of Signature Checking)
If $\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}$, then $\Gamma \vdash \mathrm{M} \Leftarrow_\kappa \mathrm{S}$.

Proof:

- Case: Rules 122 and 123. Trivial.
- Case: Rule 124.
  1. By induction, $\Gamma \vdash \mathrm{M} \Rightarrow_{\kappa'} \mathrm{R}$, where $\Gamma \vdash \mathrm{R} \leq \mathrm{S}$ and $\kappa' \sqsubseteq \kappa$.
  2. Thus, $\Gamma \vdash \mathsf{roll}(\mathrm{M}) \Rightarrow_{\kappa'} \rho\,(\mathrm{R})$, and by Corollary 6.3.1, $\Gamma \vdash \rho\,(\mathrm{R}) \leq \rho\,(\mathrm{S})$.
- Case: Rule 125.
  1. By induction, $\Gamma \vdash \mathbb{M} \Rightarrow_{\mathsf{P}} \rho\,\mathrm{X}.\mathbb{R}$ and $\Gamma \vdash \rho\,\mathrm{X}.\mathbb{R} \leq \rho\,\mathrm{X}.\mathrm{S}$.
  2. Thus, $\Gamma \vdash \mathsf{unroll}(\mathbb{M}) \Rightarrow_{\mathsf{P}} \mathbb{R}[\mathbb{M}/\mathrm{X}]$.
  3. By Soundness and Proposition 4.1.3, $\Gamma \vdash \mathsf{Fst}(\mathbb{M}) : \mathsf{Fst}(\mathbb{R})$.
  4. By inversion on subtyping, $\Gamma \vdash \mathsf{Fst}(\mathbb{R}) \leq \mathsf{Fst}(\mathrm{S})$ and $\Gamma, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}) \vdash \mathfrak{S}_{\mathbb{R}}(\mathrm{X}^{\mathsf{c}}) \leq \mathfrak{S}_{\mathrm{S}}(\mathrm{X}^{\mathsf{c}})$.
  5. By Proposition 4.1.9, $\Gamma, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}) \vdash \mathbb{R} \equiv \mathfrak{S}_{\mathbb{R}}(\mathrm{X}^{\mathsf{c}})$ and $\Gamma, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}) \vdash \mathfrak{S}_{\mathrm{S}}(\mathrm{X}^{\mathsf{c}}) \leq \mathrm{S}$.
  6. By Transitivity, $\Gamma, \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathbb{R}) \vdash \mathbb{R} \leq \mathrm{S}$.
  7. By Substitution, $\Gamma \vdash \mathbb{R}[\mathbb{M}/\mathrm{X}] \leq \mathrm{S}[\mathbb{M}/\mathrm{X}]$.

■

Finally, Figure 6.17 gives the phase-splitting rules for the new module constructs. The rules for $\mathsf{roll}(\mathrm{M})$ and $\mathsf{unroll}(\mathrm{M})$ show that these are merely retyping operations, which have no effect on either the static or dynamic part of M. The rules for $\mathsf{fetch}(\mathrm{M})$ show that the only part of M that actually needs to be fetched from memory is its dynamic part. The rule for recursive modules illustrates that the recursion only applies to the dynamic part of the recursive module body, since the static part is fully specified by the transparent declared signature. It is easy to check that these new cases preserve the soundness of phase-splitting (Proposition 4.2.7).

$\boxed{\textbf{Pure module phase-splitting: } \Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]}$

$$\frac{\Gamma \vdash \mathbb{S} \rightrightarrows [\![\mathrm{X}^{\mathsf{c}}{:}\mathbb{K}.\mathrm{C}]\!] \quad \Gamma, \mathrm{X}{:}\mathsf{maybe}(\mathbb{S}) \vdash \mathrm{M} \Rightarrow_{\mathtt{P}} \mathrm{R} \rightrightarrows [\mathrm{D}, e] \quad \Gamma, \mathrm{X}{:}\mathsf{maybe}(\mathbb{S}) \vdash \mathrm{R} \leq \mathbb{S}}{\Gamma \vdash \mathsf{rec}(\mathrm{X} : \mathbb{S}.\, \mathrm{M}) \Rightarrow_{\mathtt{P}} \mathbb{S} \rightrightarrows [\mathsf{Can}(\mathbb{K}), \mathsf{let}\ \mathrm{X}^{\mathsf{c}} = \mathsf{Can}(\mathbb{K})\ \mathsf{in}\ \mathsf{rec}(\mathrm{X}^{\mathsf{r}} : \mathrm{C}.\, e)]}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{P}} \mathsf{maybe}(\mathrm{S}) \rightrightarrows [\mathrm{C}, e]}{\Gamma \vdash \mathsf{fetch}(\mathrm{M}) \Rightarrow_{\mathtt{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, \mathsf{fetch}(e)]}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{P}} \mathrm{S} \rightrightarrows [\mathrm{C}, e]}{\Gamma \vdash \mathsf{roll}(\mathrm{M}) \Rightarrow_{\mathtt{P}} \rho\,(\mathrm{S}) \rightrightarrows [\mathrm{C}, e]} \qquad \frac{\Gamma \vdash \mathbb{M} \Rightarrow_{\mathtt{P}} \rho\,\mathrm{X}.\mathrm{S} \rightrightarrows [\mathrm{C}, e]}{\Gamma \vdash \mathsf{unroll}(\mathbb{M}) \Rightarrow_{\mathtt{P}} \mathrm{S}[\mathbb{M}/\mathrm{X}] \rightrightarrows [\mathrm{C}, e]}$$

$\boxed{\textbf{Impure module packaging: } \Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{I}} \mathrm{S} \rightrightarrows e}$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{I}} \mathsf{maybe}(\mathrm{S}) \rightrightarrows e}{\Gamma \vdash \mathsf{fetch}(\mathrm{M}) \Rightarrow_{\mathtt{I}} \mathrm{S} \rightrightarrows \mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ e\ \mathsf{in}\ \mathsf{pack}\ [\alpha, \mathsf{fetch}(x)]\ \mathsf{as}\ \langle\!|\mathrm{S}|\!\rangle}$$

$$\frac{\Gamma \vdash \mathrm{M} \Rightarrow_{\mathtt{I}} \mathrm{S} \rightrightarrows e}{\Gamma \vdash \mathsf{roll}(\mathrm{M}) \Rightarrow_{\mathtt{I}} \rho\,(\mathrm{S}) \rightrightarrows e}$$

Figure 6.17: New Module Phase-Splitting Rules

Chapter 7

Safe Recursion

I argued in Section 5.2.1 that the backpatching semantics for recursive modules is superior to a fixed-point semantics, because it ensures that any computational effects in the recursive module body will only happen once and not be repeated at each use of the recursive module variable. However, in the recursive module proposal that I sketched in Section 5.4 (and that I will formalize in Part III), the typechecker makes no attempt to detect statically whether or not recursion is *safe*.[1] That is, given a recursive module $\mathsf{rec}(\mathrm{X}:\mathrm{S}.\,\mathrm{M})$, the typechecker cannot guarantee that the evaluation of M will not prompt the dereferencing of X before it has been backpatched. As a result, whenever X is dereferenced (by $\mathsf{fetch}(\mathrm{X})$), there is a possibility that it has not yet been backpatched, in which case a run-time error must be raised.

While dynamic detection of unsafe recursion has the benefit of simplicity, compile-time detection would be preferable if it could be done in a manner that is not overly conservative. Furthermore, statically ensuring safe recursion would allow recursive modules to be implemented more efficiently. In the absence of static detection, there are two well-known implementation choices: 1) the recursive variable X can be implemented as a pointer to a value of `option` type (initially `NONE`), in which case every dereference of X must also perform a tag check to see if it has been backpatched yet, or 2) X can be implemented as a pointer to a thunk (initially `fn () => raise Error`), in which case dereferencing X does not require a tag check but instead incurs a function call. Either way, mutually recursive functions defined across module boundaries may be noticeably slower than ordinary ML functions. If recursion is statically determined to be safe, though, the value pointed to by X will be needed only after X has been backpatched, so each $\mathsf{fetch}(\mathrm{X})$ will require only a single pointer dereference.

In this chapter, I consider the problem of statically detecting whether recursion under a backpatching semantics is safe. In order to simplify matters and isolate orthogonal concerns, I will ignore the issues involving type components in recursive modules and focus attention on their dynamic components. In particular, I will consider the semantics of a safe recursive *term* construct $\mathsf{saferec}(x:\mathrm{C}.\,e)$ in the context of the simply-typed λ-calculus. Even in the limited setting of the simply-typed λ-calculus, the safe recursion problem is quite interesting and difficult, and there has been little previous work on it.

The chapter is structured as follows: In Section 7.1, I introduce the property of *evaluability*, which guarantees that a term is safe to evaluate even if it has free references to undefined recursive variables. I consider a simple straw-man approach to tracking evaluability, and illustrate by

[1] In Dreyer [10], I used the term *well-founded recursion* instead of *safe recursion*. Both terms appear in the literature, but the former gives the impression of similarity to *well-founded induction* when there is none. I have therefore opted to use the latter term in this chapter.

examples why a more sophisticated approach is required.

In Section 7.2, I propose a type-theoretic approach to resolving these problems. The basic idea is to model recursive variables statically as *names*, and to use names to track the set of recursive variables that a piece of code may attempt to dereference when evaluated.[2] Names are useful for two purposes: (1) tracking uses of multiple recursive variables in the presence of nested recursion, and (2) supporting separate compilation of safe recursive terms/modules. An equally important feature of my approach is that recursive terms/modules may invoke "legacy" functions defined in existing ML code without requiring them to be changed or recompiled to account for name reasoning. Nevertheless, as I discuss in Section 7.2.3, there are useful recursive module idioms for which instrumentation of existing ML code appears to be unavoidable if one wants to statically ensure that the recursion is safe.

Interestingly, while computational effects are what necessitate the backpatching semantics of recursion, all of the subtleties involving names are explored in Section 7.2 in the setting of the pure λ-calculus. Section 7.3 introduces computational effects into the language in the form of mutable state and continuations, but these extensions turn out to be rather simple, orthogonal, and oblivious to the presence of names.

In Section 7.4, I show how the unrestricted recursive term construct $\mathsf{rec}(x : \mathrm{C}.\, e)$ (whose semantics was formalized in Section 6.2 of the previous chapter) may be encoded in terms of the safe recursive construct $\mathsf{saferec}(x : \mathrm{C}.\, e)$ by extending the language with *memoized computations*. While the unrestricted construct does not statically ensure safe recursion, it is useful to have as a fallback in circumstances where the type system is too weak to observe that a safe recursive term is in fact safe.

In Section 7.5, I compare my approach with related work on safe recursion and backpatching semantics. Finally, in Section 7.6, I discuss the key issues and difficulties I foresee in scaling my approach to the level of a realistic ML extension.

7.1 Evaluability

Recall the recursive construct $\mathsf{rec}(x : \mathrm{C}.\, e)$ introduced in Section 6.2. The typing rule for $\mathsf{rec}(x : \mathrm{C}.\, e)$ checks that e has type C in a context where x has type $\mathsf{maybe}(\mathrm{C})$. To dereference x, we write $\mathsf{fetch}(x)$.

Now consider a "safe" recursive construct of the form $\mathsf{saferec}(x : \mathrm{C}.\, e)$. What must we require of e to ensure that $\mathsf{saferec}(x : \mathrm{C}.\, e)$ is in fact safe? Crary *et al.* [6] require that e be *valuable* (that is, pure and terminating) in a context where x is not. Let us generalize their notion of valuability to one permitting effects, which I will call *evaluability*. A term may be judged "evaluable" if its evaluation does not dereference any undefined (*i.e.,* unbackpatched) recursive variable. Thus, to ensure $\mathsf{saferec}(x : \mathrm{C}.\, e)$ is safe, the expression e must be evaluable in a context where dereferences of the variable x are considered non-evaluable. A term can be non-evaluable and still be well-formed, but only evaluable expressions are safe to evaluate in the presence of undefined recursive variables.

Formally, we might incorporate evaluability into the type system by dividing the typing judgment into one classifying evaluable terms ($\Gamma \vdash e \downarrow \mathrm{C}$) and one classifying non-evaluable terms ($\Gamma \vdash e \uparrow \mathrm{C}$). (There is an implicit inclusion of the former in the latter.) The typing rule for

[2] My use of names is inspired by the work of Nanevski on a core language for metaprogramming and symbolic computation [57], although it is closer in detail to his work (concurrent with mine) on using names to model control effects [58].

$\mathsf{saferec}(x : \mathrm{C}.\, e)$ might then be as follows:

$$\frac{\Gamma, x{:}\mathsf{box}(\mathrm{C}) \vdash e \downarrow \mathrm{C}}{\Gamma \vdash \mathsf{saferec}(x : \mathrm{C}.\, e) \downarrow \mathrm{C}}$$

To stress the distinction between safe and unsafe recursion, I have chosen here to bind x with a new type, $\mathsf{box}(\mathrm{C})$, instead of $\mathsf{maybe}(\mathrm{C})$. To dereference a location v of type $\mathsf{box}(\mathrm{C})$, we will write $\mathsf{unbox}(v)$.

It is important to understand that the distinction between $\mathsf{box}(\mathrm{C})$ and $\mathsf{maybe}(\mathrm{C})$, and between unbox'ing and fetch'ing, is not merely pedantic. The implementation of $\mathsf{fetch}(v)$ must check to make sure that v has been backpatched, whereas the implementation of $\mathsf{unbox}(v)$ need not perform any such check—the static semantics will guarantee that v will not be unbox'ed until it has been backpatched. To validate this guarantee, the typing rule for $\mathsf{unbox}(v)$ must be conservative and treat all unboxing operations as potentially non-evaluable:

$$\frac{\Gamma \vdash v \downarrow \mathsf{box}(\mathrm{C})}{\Gamma \vdash \mathsf{unbox}(v) \uparrow \mathrm{C}}$$

7.1.1 The Evaluability Judgment

While true evaluability is clearly an undecidable property, there are certain kinds of expressions that we can expect the type system to recognize as evaluable. Certainly all values and projections from values should be considered evaluable, as should all let-expressions whose constituent expressions are evaluable. There is no reason to limit evaluability to pure expressions either. For instance, the ML expressions `ref(`e`)`, `!`e, and e_1`:=`e_2 should all be evaluable as long as their constituent expressions are. Given this characterization, we can already observe that the effectful recursive module example in Figure 5.6 from Chapter 5 is safe. Evaluability is thus independent of computational purity in the traditional sense.

There is, however, a correspondence between *non*-evaluability and computational *im*-purity in the sense that both are hidden by λ-abstractions and unleashed by function applications. In ML we assume (for the purpose of the value restriction on let-polymorphism) that all function applications are potentially impure. In the current setting we might similarly assume for simplicity that all function applications are potentially non-evaluable.

Unfortunately, this assumption has one major drawback: it implies that we can never evaluate a function application inside the body of a recursive term! Furthermore, it is usually unnecessary: while functions defined inside a recursive term may very well be concealing references to an undefined recursive variable, functions defined in existing ML code will not. For example, suppose we were to modify the example of Figure 5.6 in the manner shown in Figure 7.1. Instead of defining `A.debug` as a boolean flag (`ref false`), the new version defines it as a mutable array, by a call to the array creation function `Array.array`. The call to `Array.array` is perfectly evaluable. In contrast, a call to the function `A.f` from within the recursive module would *not* be, since the body of `A.f` makes a recursive call to `unbox(X).B.g`. Lumping them together and assuming the worst makes the evaluability judgment overly conservative.

7.1.2 A Total/Partial Distinction

At the very least, then, we should make a type distinction between functions whose bodies are evaluable and those whose bodies are not. Thinking of non-evaluability as a sort of computational effect, let us refer to the first type of function as "total" (written $\mathrm{C}_1 \xrightarrow{\mathsf{tot}} \mathrm{C}_2$) and to the latter type

```
saferec (X : SIG.
struct
  structure A = struct
    val debug = Array.array(n,0)
    fun f(x) = ...unbox(X).B.g(x+1)...
    ...
  end
  structure B = struct ... end
end)
```

Figure 7.1: Modified Example of Recursive Module With Effects

as "partial" (written $\mathrm{C}_1 \xrightarrow{\mathsf{par}} \mathrm{C}_2$).[3] The typing rules for total and partial λ-abstractions would then be as follows:

$$\frac{\Gamma, x : \mathrm{D} \vdash e \downarrow \mathrm{C}}{\Gamma \vdash \lambda\, x : \mathrm{D}.\, e \downarrow \mathrm{D} \xrightarrow{\mathsf{tot}} \mathrm{C}} \qquad \frac{\Gamma, x : \mathrm{D} \vdash e \uparrow \mathrm{C}}{\Gamma \vdash \lambda\, x : \mathrm{D}.\, e \downarrow \mathrm{D} \xrightarrow{\mathsf{par}} \mathrm{C}}$$

Correspondingly, applications of total functions will be deemed evaluable, whereas applications of partial functions will be assumed non-evaluable:

$$\frac{\Gamma \vdash v_1 \downarrow \mathrm{D} \xrightarrow{\mathsf{tot}} \mathrm{C} \quad \Gamma \vdash v_2 \downarrow \mathrm{D}}{\Gamma \vdash v_1(v_2) \downarrow \mathrm{C}} \qquad \frac{\Gamma \vdash v_1 \downarrow \mathrm{D} \xrightarrow{\mathsf{par}} \mathrm{C} \quad \Gamma \vdash v_2 \downarrow \mathrm{D}}{\Gamma \vdash v_1(v_2) \uparrow \mathrm{C}}$$

The total/partial distinction addresses the concerns discussed in the previous section, to an extent. Existing ML functions can now be classified as total, and the arrow type C_1 `->` C_2 in ML proper is synonymous with a total arrow. Thus, we may now evaluate calls to existing ML functions in the presence of undefined recursive variables, as those function applications will be known to be evaluable. Nonetheless, some serious problems remain.

7.1.3 Limitations of the Total/Partial Distinction

First, consider what happens if we use the safe recursive construct to define a single recursive function, such as factorial:

$$\texttt{saferec(f : int} \xrightarrow{\mathsf{par}} \texttt{int. fn x => ... x * unbox(f)(x-1) ...)}$$

Note that we are forced to give the recursive expression a partial arrow type because the body of the factorial function unboxes the recursive variable `f`. Nonetheless, exporting factorial as a partial function is bad because it means that no application of factorial can ever be evaluated inside another recursive expression!

To mend this problem, we observe that while the factorial function is indeed partial *during* the evaluation of the general recursive expression defining it, it becomes total as soon as `f` is backpatched with a definition. One way to incorporate this observation into the type system is to revise the typing rule for recursive terms $\mathsf{saferec}(x : \mathrm{C}.\, e)$ so that we ignore partial/total discrepancies when matching the declared type C with the actual type of e. For example, in the factorial definition

[3]Note: This total/partial distinction is completely unrelated to the total/partial distinction on *functors* from Chapter 4, which corresponds to applicative vs. generative behavior.

above, we would allow `f` to be declared with a total arrow $\mathtt{int} \xrightarrow{\mathsf{tot}} \mathtt{int}$, since the body of the definition has an equivalent type *modulo* a partial/total mismatch.

Unfortunately, such a revised typing rule is only sound if we prohibit nesting of recursive terms. Otherwise, the rule may allow a truly partial function to be erroneously assigned a total type, as the following code illustrates:

```
saferec (x : C.
  let
     val f = saferec(y : unit -tot-> C. fn () => unbox(x))
  in
     f()
  end
)
```

The trouble here is that the evaluation of the recursive term defining `f` results only in the back-patching of `y`, not `x`. It is therefore unsound for that term to make the type of `fn () => unbox(x)` total. In short, the problem is that the total/partial dichotomy is too coarse because it does not distinguish between the dereferencing of different recursive variables. In the type system of Section 7.2, we will be able to give `f` a more appropriate type specifying that `f` will dereference `x` when applied, but not `y`.

Another problem with the total/partial distinction arises in the use of higher-order functions. Suppose we wish to use the Standard Basis `map` function for lists, which can be given the following type (for any D and C):

$$\mathtt{val\ map\ :\ } (\mathrm{D} \xrightarrow{\mathsf{tot}} \mathrm{C}) \xrightarrow{\mathsf{tot}} (\mathrm{D}\ \mathtt{list} \xrightarrow{\mathsf{tot}} \mathrm{C}\ \mathtt{list})$$

Since the type of `map` is a pure ML type, all the arrows are total, which means that we cannot apply `map` to a partial function, as in the following:

```
saferec (X : SIG.
  let
     val f : D -par-> C = ...
     val g : D list -par-> C list = map f
  ...
)
```

Given the type of `map`, this is reasonable: unless we know how `map` is implemented, we have no way of knowing that evaluating `map f` will not try to apply `f`, resulting in a potential dereference of `X`.

Nevertheless, we should at least be able to replace `map f` with `fn xs => map f xs`, its eta-expansion, which is clearly evaluable since it is a value. Even its eta-expansion is ill-typed, however, because the type of `f` still does not match the argument type of `map`. The way I propose to resolve this problem is to view a partial/total type mismatch not as a sign that the offending expression (in this case, `map f`) is ill-typed, but merely that it is potentially non-evaluable. The type system of Section 7.2 will reflect this intuition, and will correspondingly consider the function `fn xs => map f xs` to be well-typed with a *partial* arrow, but not a total one.

7.2 A Type System for Safe Recursion

In this section I present a type system for safe recursion that addresses both of the problems enumerated in the previous section. To address the nested recursion problem, I generalize the

$$
\begin{array}{lrcl}
\text{Variables} & x, y, z & \in & \mathit{Variables} \\
\text{Names} & \mathcal{X}, \mathcal{Y}, \mathrm{Z} & \in & \mathit{Names} \\
\text{Supports} & \mathcal{S}, \mathcal{T} & \in & \mathcal{P}_{\mathrm{fin}}(\mathit{Names}) \\
\text{Types} & \mathrm{C}, \mathrm{D} & ::= & \mathsf{unit} \mid \mathrm{C}_1 \times \mathrm{C}_2 \mid \mathrm{C}_1 \xrightarrow{\mathcal{S}} \mathrm{C}_2 \mid \forall \mathcal{X}.\mathrm{C} \mid \mathsf{box}_{\mathcal{S}}(\mathrm{C}) \\
\text{Values} & v & ::= & x \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid \lambda^{\mathcal{S}} x : \mathrm{C}.\, e \mid \lambda \mathcal{X}.e \\
\text{Terms} & e, f & ::= & v \mid \pi_i(v) \mid v_1(v_2) \mid v(\mathcal{S}) \mid \mathsf{box}_{\mathcal{S}}(v) \mid \mathsf{unbox}(v) \mid \\
 & & & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e) \\
\text{Typing Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : \mathrm{C} \mid \Gamma, \mathcal{X}
\end{array}
$$

Figure 7.2: Syntax of Safe Recursion Language

judgment of evaluability to one that tracks uses of individual recursive variables. I achieve this by introducing along with each recursive variable x a *name* $\mathcal{X}$ that is used as a static representative of the variable. The new evaluability judgment has the form $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]$, with the interpretation "under context Γ, term e has type C and is evaluable *modulo* the names in set $\mathcal{S}$". In other words, e will evaluate without dereferencing any recursive variables *except* possibly those whose associated names appear in $\mathcal{S}$. Following Nanevski [57], I call a finite set of names a *support*. Our previous judgment of evaluability ($\Gamma \vdash e \downarrow \mathrm{C}$) can be understood as evaluability modulo the empty support ($\Gamma \vdash e : \mathrm{C}\ [\emptyset]$), while non-evaluability ($\Gamma \vdash e \uparrow \mathrm{C}$) corresponds to evaluability modulo *some* non-empty support.

Similarly, I will generalize the types of functions to bear a support indicating which particular recursive variables may be dereferenced in their bodies. Thus, the total arrow type of the previous section becomes an arrow type bearing empty support, while the partial arrow type corresponds to an arrow type bearing *some* non-empty support.

To address the higher-order function problem, I employ a novel judgment of *type equivalence modulo a support*, which allows type mismatches in an expression to be ignored so long as they only involve names that are in the support of the expression. The intuition behind this judgment is as follows. If a name $\mathcal{X}$ is in the support of an expression e, then the evaluation of e may dereference the recursive variable x to which $\mathcal{X}$ corresponds. The type system must therefore ensure that x is backpatched before e is ever evaluated. Once x is backpatched, however, the effect of dereferencing it becomes *benign*, and the name $\mathcal{X}$ can subsequently be ignored. Thus, since e will only be evaluated after x has been backpatched, type mismatches regarding $\mathcal{X}$ are irrelevant as far as the typechecking of e is concerned.

7.2.1 Syntax

The syntax of the language is given in Figure 7.2. I assume the existence of countably infinite sets of names (*Names*) and variables (*Variables*), and use $\mathcal{S}$ and $\mathcal{T}$ to range over supports. I will sometimes write the name $\mathcal{X}$ as shorthand for the singleton support $\{\mathcal{X}\}$.

The type structure of the language is as follows. Unit (unit) and pair types ($\mathrm{C}_1 \times \mathrm{C}_2$) require no explanation. An arrow type ($\mathrm{C}_1 \xrightarrow{\mathcal{S}} \mathrm{C}_2$) bears a support $\mathcal{S}$ on the arrow, which indicates the names whose associated recursive variables must be backpatched before a function of this type may be applied. Similarly, λ-abstractions $\lambda^{\mathcal{T}} x : \mathrm{D}.\, e$ explicitly specify the support $\mathcal{T}$ required for them to be applied. I will write $\mathrm{C}_1 \to \mathrm{C}_2$ (resp. $\lambda\, x : \mathrm{D}.\, e$) as shorthand for $\mathrm{C}_1 \xrightarrow{\emptyset} \mathrm{C}_2$ (resp. $\lambda^{\emptyset} x : \mathrm{D}.\, e$).

The language also provides the ability to abstract a term over a name. The type $\forall\mathcal{X}.\mathrm{C}$ classifies name abstraction values $\lambda\mathcal{X}.e$. Application of a name abstraction, $v(\mathcal{S})$, allows the name parameter of v to be instantiated with a support $\mathcal{S}$, not just a single name. The reasons for allowing names to be instantiated with supports are discussed in Section 7.2.3.

Lastly, the location type $\mathsf{box}_{\mathcal{S}}(\mathrm{C})$ classifies a memory location that will contain a value of type C once the recursive variables associated with the names in $\mathcal{S}$ have been backpatched. Locations are most commonly introduced by recursive terms. The recursive term construct $\mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e)$ binds both the name $\mathcal{X}$ and the variable x in e. The type of x will be $\mathsf{box}_{\mathcal{X}}(\mathrm{C})$, indicating that x may only be unboxed by an expression with support containing $\mathcal{X}$. The $\mathsf{box}_{\mathcal{S}}(\mathrm{C})$ type may also be introduced by $\mathsf{box}_{\mathcal{S}}(v)$, which creates a new memory location and stores v at it. As this is an effectful operation, $\mathsf{box}_{\mathcal{S}}(v)$ is not a value—the only values of box type are variables. The elimination form for box types is $\mathsf{unbox}(v)$, which dereferences the location v. I will sometimes write $\mathsf{box}(\mathrm{C})$ (resp. $\mathsf{box}(v)$) as shorthand for $\mathsf{box}_{\emptyset}(\mathrm{C})$ (resp. $\mathsf{box}_{\emptyset}(v)$).

For notational convenience, I will enforce several implicit requirements on the syntactic well-formedness of contexts and judgments. A context Γ is well-formed if (1) it does not bind the same variable/name twice, and (2) for any prefix of Γ of the form $\Gamma', x : \mathrm{C}$, the free names of C are bound in Γ'. A judgment of the form $\Gamma \vdash \mathcal{J}$ is well-formed if (1) Γ is well-formed, and (2) any free names appearing in $\mathcal{J}$ are bound in Γ. I assume and maintain the implicit invariant that all contexts and judgments are well-formed.

7.2.2 Static Semantics

The main typing judgment has the form $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]$. The support $\mathcal{S}$ represents the set of names whose associated recursive variables we may assume have been backpatched by the time e is evaluated. Put another way, the only recursive variables that e may dereference are those associated with the names in $\mathcal{S}$. The static semantics is carefully designed to validate this assumption.

The rules of the type system, shown in Figure 7.3, are designed to make admissible the principle of *support weakening*, which says that if $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]$ then $\Gamma \vdash e : \mathrm{C}\ [\mathcal{T}]$ for any $\mathcal{T} \supseteq \mathcal{S}$. Thus, for instance, since a variable x does not require any support, Rule 1 allows x to be assigned any support $\mathcal{S} \subseteq \mathrm{dom}(\Gamma)$, not just the empty support.

The remainder of the rules may be summarized as follows. Unit, pairs and projections need no support (Rules 2, 3 and 4). A function $\lambda^{\mathcal{T}} x : \mathrm{D}.\, e$ has type $\mathrm{D} \xrightarrow{\mathcal{T}} \mathrm{C}$ in any support $\mathcal{S}$, so long as the body e is well-typed under the addition of support $\mathcal{T}$ (Rule 5). To evaluate a function application $v_1(v_2)$, the support $\mathcal{S}$ must contain the support $\mathcal{T}$ on v_1's arrow type (Rule 6).

Although a name abstraction $\lambda\mathcal{X}.e$ suspends the evaluation of e, the body is typechecked under the same support as the abstraction itself (Rule 7). In other words, one can view $\forall\mathcal{X}.\mathrm{C}$ as another kind of arrow type that always bears empty support (compare with Rule 5 when $\mathcal{T} = \emptyset$). Note also that the assumptions about the well-formedness of judgments ensure that the support $\mathcal{S}$ cannot contain $\mathcal{X}$, since $\mathcal{S} \subseteq \mathrm{dom}(\Gamma)$ and $\mathcal{X} \notin \mathrm{dom}(\Gamma)$. Restricting name abstractions in this way is motivated by the fact that, in all my intended uses of name abstractions (see Section 7.2.3 below), the body of the abstraction is a value (with empty support).

Instantiating a name abstraction v of type $\forall\mathcal{X}.\mathrm{C}$ with a support $\mathcal{T}$ has the type resulting from substituting $\mathcal{T}$ for $\mathcal{X}$ in C (Rule 8). The substitution $\mathrm{C}[\mathcal{T}/\mathcal{X}]$ is defined by replacing every support $\mathcal{S}$ appearing in C with $\mathcal{S}[\mathcal{T}/\mathcal{X}]$, which is in turn defined as follows:

$$\mathcal{S}[\mathcal{T}/\mathcal{X}] \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathcal{S} \cup \mathcal{T} - \{\mathcal{X}\} & \text{if } \mathcal{X} \in \mathcal{S} \\ \mathcal{S} & \text{if } \mathcal{X} \notin \mathcal{S} \end{cases}$$

Well-formed terms: $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]$

$$\frac{x : \mathrm{C} \in \Gamma}{\Gamma \vdash x : \mathrm{C}\ [\mathcal{S}]}\ (1) \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathsf{unit}\ [\mathcal{S}]}\ (2)$$

$$\frac{\Gamma \vdash v_1 : \mathrm{C}_1\ [\mathcal{S}] \quad \Gamma \vdash v_2 : \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \langle v_1, v_2\rangle : \mathrm{C}_1 \times \mathrm{C}_2\ [\mathcal{S}]}\ (3) \qquad \frac{\Gamma \vdash v : \mathrm{C}_1 \times \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \pi_i(v) : \mathrm{C}_i\ [\mathcal{S}]}\ (4)$$

$$\frac{\Gamma, x : \mathrm{D} \vdash e : \mathrm{C}\ [\mathcal{S} \cup \mathcal{T}]}{\Gamma \vdash \lambda^{\mathcal{T}} x : \mathrm{D}.\, e : \mathrm{D} \xrightarrow{\mathcal{T}} \mathrm{C}\ [\mathcal{S}]}\ (5) \qquad \frac{\Gamma \vdash v_1 : \mathrm{D} \xrightarrow{\mathcal{T}} \mathrm{C}\ [\mathcal{S}] \quad \Gamma \vdash v_2 : \mathrm{D}\ [\mathcal{S}] \quad \mathcal{T} \subseteq \mathcal{S}}{\Gamma \vdash v_1(v_2) : \mathrm{C}\ [\mathcal{S}]}\ (6)$$

$$\frac{\Gamma, \mathcal{X} \vdash e : \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \lambda\mathcal{X}.e : \forall\mathcal{X}.\mathrm{C}\ [\mathcal{S}]}\ (7) \qquad \frac{\Gamma \vdash v : \forall\mathcal{X}.\mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash v(\mathcal{T}) : \mathrm{C}[\mathcal{T}/\mathcal{X}]\ [\mathcal{S}]}\ (8)$$

$$\frac{\Gamma \vdash v : \mathrm{C}\ [\mathcal{S} \cup \mathcal{T}]}{\Gamma \vdash \mathsf{box}_{\mathcal{T}}(v) : \mathsf{box}_{\mathcal{T}}(\mathrm{C})\ [\mathcal{S}]}\ (9) \qquad \frac{\Gamma \vdash v : \mathsf{box}_{\mathcal{T}}(\mathrm{C})\ [\mathcal{S}] \quad \mathcal{T} \subseteq \mathcal{S}}{\Gamma \vdash \mathsf{unbox}(v) : \mathrm{C}\ [\mathcal{S}]}\ (10)$$

$$\frac{\Gamma \vdash e_1 : \mathrm{D}\ [\mathcal{S}] \quad \Gamma, x : \mathrm{D} \vdash e_2 : \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \mathrm{C}\ [\mathcal{S}]}\ (11)$$

$$\frac{\Gamma, \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}) \vdash e : \mathrm{D}\ [\mathcal{S}] \quad \Gamma, \mathcal{X} \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{X}]}{\Gamma \vdash \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e) : \mathrm{C}\ [\mathcal{S}]}\ (12)$$

$$\frac{\Gamma \vdash e : \mathrm{D}\ [\mathcal{S}] \quad \Gamma \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]}\ (13)$$

Type equivalence: $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]$

$$\frac{}{\Gamma \vdash \mathsf{unit} \equiv \mathsf{unit}\ [\mathcal{S}]}\ (14) \qquad \frac{\Gamma \vdash \mathrm{D}_1 \equiv \mathrm{D}_2\ [\mathcal{S}] \quad \Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \mathrm{D}_1 \times \mathrm{C}_1 \equiv \mathrm{D}_2 \times \mathrm{C}_2\ [\mathcal{S}]}\ (15)$$

$$\frac{\mathcal{S} \cup \mathcal{S}_1 = \mathcal{T} = \mathcal{S} \cup \mathcal{S}_2 \quad \Gamma \vdash \mathrm{D}_1 \equiv \mathrm{D}_2\ [\mathcal{T}] \quad \Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{T}]}{\Gamma \vdash \mathrm{D}_1 \xrightarrow{\mathcal{S}_1} \mathrm{C}_1 \equiv \mathrm{D}_2 \xrightarrow{\mathcal{S}_2} \mathrm{C}_2\ [\mathcal{S}]}\ (16)$$

$$\frac{\Gamma, \mathcal{X} \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \forall\mathcal{X}.\mathrm{C}_1 \equiv \forall\mathcal{X}.\mathrm{C}_2\ [\mathcal{S}]}\ (17) \qquad \frac{\mathcal{S} \cup \mathcal{S}_1 = \mathcal{T} = \mathcal{S} \cup \mathcal{S}_2 \quad \Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{T}]}{\Gamma \vdash \mathsf{box}_{\mathcal{S}_1}(\mathrm{C}_1) \equiv \mathsf{box}_{\mathcal{S}_2}(\mathrm{C}_2)\ [\mathcal{S}]}\ (18)$$

Figure 7.3: Static Semantics for Safe Recursion Language

Boxing a value requires no support (Rule 9). Unboxing a value v of type $\mathsf{box}_{\mathcal{T}}(\mathrm{C})$ is only permitted if the recursive variables associated with the names in $\mathcal{T}$ have been defined, *i.e.,* if $\mathcal{T}$ is contained in the support $\mathcal{S}$ (Rule 10). Let-terms have the support of their constituent expressions.

Rules 12 and 13 are the most interesting rules in the type system since they both make use of the judgment of type equivalence modulo a support, also defined in Figure 7.3. The judgment $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]$ means that C_1 and C_2 are equivalent types *modulo* the names in support $\mathcal{S}$, *i.e.,* that C_1 are C_2 are identical types if we ignore all occurrences of the names in $\mathcal{S}$. For example, the types $\mathrm{D} \xrightarrow{\emptyset} \mathrm{C}$ and $\mathrm{D} \xrightarrow{\mathcal{X}} \mathrm{C}$ are equivalent *modulo* any support containing $\mathcal{X}$. In addition, when checking equivalence of arrow types $\mathrm{D}_1 \xrightarrow{\mathcal{S}_1} \mathrm{C}_1$ and $\mathrm{D}_2 \xrightarrow{\mathcal{S}_2} \mathrm{C}_2$ modulo $\mathcal{S}$, we compare the argument types and result types at the extended modulus $\mathcal{S} \cup \mathcal{S}_1 = \mathcal{S} \cup \mathcal{S}_2$ instead of $\mathcal{S}$. This makes sense because a function of one of these types may only be applied with $\mathcal{S} \cup \mathcal{S}_1$ in the support. The rule for box can be justified similarly.

This notion of equivalence modulo a support is critical to the typing of recursive terms (Rule 12). Recall the factorial example from Section 7.1.2, adapted to the present type system:

$$\texttt{saferec}\ (\mathcal{X} \triangleright \texttt{f : int} \xrightarrow{\emptyset} \texttt{int.}$$
$$\texttt{fn x => ... x * unbox(f)(x-1) ...)}$$

The issue here is that the declared type of `f` does not match the type of the body, $\texttt{int} \xrightarrow{\mathcal{X}} \texttt{int}$. Once `f` is backpatched, however, the two types do match *modulo* $\mathcal{X}$.

Correspondingly, the typing rule for recursive terms $\mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e)$ works as follows. First, the context Γ is extended with the name $\mathcal{X}$, as well as the recursive variable x of type $\mathsf{box}_{\mathcal{X}}(\mathrm{C})$. This location type binds the name and the variable together because it says that $\mathcal{X}$ must be in the support of any expression that attempts to dereference (unbox) x. The rule then checks that e has some type D in this extended context, under a support $\mathcal{S}$ that does *not* include $\mathcal{X}$ (since x is undefined while evaluating e). Finally, it checks that D and C are equivalent *modulo* $\mathcal{X}$. It is easiest to understand this last step as a generalization of our earlier idea of ignoring discrepancies between partial and total arrows when comparing D and C. The difference here is that we ignore discrepancies with respect to a particular name $\mathcal{X}$ instead of all names, so that the rule behaves properly in the presence of multiple names (nested recursion).

In contrast, Rule 13 appears rather straightforward, allowing a term with type D and support $\mathcal{S}$ to be assigned a type that is equivalent to D modulo the names in $\mathcal{S}$. In fact, this rule solves the higher-order function problem described in Section 7.1.2! Recall that we wanted to apply an existing higher-order ML function like `map` to a partial function, *i.e.,* one whose arrow type bears non-empty support:

```
saferec (𝒳 ▷ x : SIG.
  let
     val f : D ──𝒳──> C = ...
     val g : D list ──𝒳──> C list = fn xs => map f xs
  ...
)
```

The problem here is that the type of `f` does not match the argument type $\mathrm{D} \xrightarrow{\emptyset} \mathrm{C}$ of `map`. Intuitively, though, this code ought to typecheck: if we are willing to add $\mathcal{X}$ to the support of `g`'s arrow type, then `x` must be backpatched before `g` is ever applied, so $\mathcal{X}$ should be ignored when typing the body of `g`.

Rule 13 encapsulates this reasoning. Since `g` is specified with type $\mathrm{D}\ \texttt{list} \xrightarrow{\mathcal{X}} \mathrm{C}\ \texttt{list}$, we can assume support $\mathcal{X}$ when typechecking its body (`map f xs`). Under support $\mathcal{X}$, Rule 13 allows us to assign `f` the type $\mathrm{D} \xrightarrow{\emptyset} \mathrm{C}$, as it is equivalent to `f`'s type *modulo* $\mathcal{X}$. Thus, `g`'s body is well-typed under support $\mathcal{X}$.

7.2.3 Separate Compilation, Non-strictness and Name Abstractions

In this section, I will give some motivation for the feature of name abstractions, $\lambda\mathcal{X}.e$. Recall that the original reason for making the unboxing (or fetching) of a recursive variable an explicit operation was to support separate compilation of recursive modules. In the separate compilation scenario (Figure 5.11) described in Section 5.2.4, the modules `Expr` and `Bind` were separately compiled as *functors* `F_Expr` and `F_Bind`, and linked together as follows:

```
F_Expr = λ𝒳. λx : box𝒳(EXPR_BIND). ...

F_Bind = λ𝒳. λx : box𝒳(EXPR_BIND). ...

saferec (𝒳 ▷ X : EXPR_BIND.
struct
  structure Expr = F_Expr{𝒳}(X)
  structure Bind = F_Bind{𝒳}(X)
end)
```

Figure 7.4: Revised Separate Compilation Scenario

```
saferec (𝒳 ▷ X : EXPR_BIND.
struct
  structure Expr = F_Expr(X)
  structure Bind = F_Bind(X)
end
```

For the purposes of this chapter, since we are ignoring the issues involving type components in modules, let us ignore the problems with using functors for separate compilation of recursive modules (for instance, the double vision problem). Instead, consider the following question: how can we ensure that this recursive module linking `Expr` and `Bind` is *safe*?

In order for the linking module to be safe, it must be the case that `F_Expr` and `F_Bind`, when applied, do not attempt to dereference their argument. That is to say, `F_Expr` and `F_Bind` must be *non-strict* functors. What types can we give to `F_Expr` and `F_Bind` to reflect the property that they are non-strict? Suppose that `F_Expr`'s return type is `EXPR`. We would like to assign it the type $\mathsf{box}_{\mathcal{X}}(\texttt{EXPR\_BIND}) \xrightarrow{\emptyset} \texttt{EXPR}$, so that (1) its argument type matches the type of `X`, and (2) the absence of $\mathcal{X}$ on the arrow indicates that `F_Expr` can be applied under empty support. However, this type makes no sense at the place where `F_Expr` is defined, because the name $\mathcal{X}$ is not in scope outside of the recursive module.

This is where name abstractions come in. To show that `F_Expr` is non-strict, it is irrelevant what particular support is required to unbox its argument, so we can use a name abstraction to allow any name or support to be substituted for `X`. Figure 7.4 shows the resulting well-typed separate compilation scenario, in which the type of `F_Expr` is $\forall\mathcal{X}.\mathsf{box}_{\mathcal{X}}(\texttt{EXPR\_BIND}) \xrightarrow{\emptyset} \texttt{EXPR}$.

The recursive term construct is still not quite as flexible for separate compilation purposes as one might like. In particular, suppose that we wanted to parameterize `F_Expr` over *just* `F_Bind` instead of *both* `F_Expr` and `F_Bind`. There is no way in our system to extract a value of type $\mathsf{box}_{\mathcal{X}}(\texttt{BIND})$ from `X` without unboxing it. It would be easy to remedy this problem, however, by generalizing the recursive construct to an n-ary one, $\mathsf{saferec}(\vec{\mathcal{X}} \triangleright \vec{x} : \vec{\mathrm{C}}.\, \vec{e})$, where each of the n recursive variables x_i is boxed separately with type $\mathsf{box}_{\mathcal{X}_i}(\mathrm{C}_i)$.

Name abstractions can also be used to express non-strictness of *general-purpose* ML functors, which in turn allows better static detection of safe recursion in certain cases. For instance, the recursive module in Figure 7.5 provides a type `C.t`, which is defined in terms of *sets* of itself. (This is a greatly simplified variant of the "bootstrapped heap" example from Section 5.1.) The definition of module `C` refers recursively to the `CSet` module, which is defined by applying the `MakeSet` functor to the `C` module. The only way we can be sure that the recursion is safe is if we know that the application of the `MakeSet` functor will not attempt to apply the partial

```
structure rec C : ORDERED = struct
  datatype t = ...CSet.set...
  fun compare (x,y) = ...CSet.compare(a,b)...
end
and CSet = MakeSet(C)
```

Figure 7.5: Recursive Module Example With Non-strict Functor Application

function `C.compare`, *i.e.,* that the `MakeSet` functor is non-strict. With name abstractions, we can instrument the implementation of `MakeSet` in order to assign it a non-strict type[4] such as $\forall\mathcal{X}.\mathsf{box}_{\mathcal{X}}(\mathtt{ORDERED}) \xrightarrow{\emptyset} \mathtt{SET}$.

Similarly, name abstractions can be used to give more precise types to core-level ML functions. For instance, suppose we had access to the code for the `map` function. By wrapping the definition of `map` in a name abstraction, we could assign the function the type

$$\forall\mathcal{X}.\ (\mathrm{D} \xrightarrow{\mathcal{X}} \mathrm{C}) \xrightarrow{\emptyset} (\mathrm{D}\ \mathtt{list} \xrightarrow{\mathcal{X}} \mathrm{C}\ \mathtt{list})$$

This type indicates that `map` will turn a value of any arrow type into a value of an arrow type bearing the same support, but will not apply its argument in the process. Given this type for `map`, we can write our recursive module example involving `map` the way we wanted to write it originally in Section 7.1.3:

$$
\begin{array}{l}
\mathtt{saferec}\ (\mathcal{X} \triangleright \mathtt{x} : \mathtt{SIG}. \\
\quad \mathtt{let} \\
\qquad \mathtt{val\ f} : \mathrm{D} \xrightarrow{\mathcal{X}} \mathrm{C} = \ldots \\
\qquad \mathtt{val\ g} : \mathrm{D}\ \mathtt{list} \xrightarrow{\mathcal{X}} \mathrm{C}\ \mathtt{list} = \boxed{\mathtt{map}\ \{\mathcal{X}\}\ \mathtt{f}} \\
\quad \ldots \\
)
\end{array}
$$

The more precise non-strict type for `map` allows us to avoid eta-expanding `map f`, but it also requires having access to the implementation of `map`. Furthermore, it requires us to modify the type of `map`, infecting the existing ML infrastructure with names. It is therefore important that, in the absence of this solution, our type system is strong enough (thanks to Rule 13) to typecheck at least the eta-expansion of `map f`, without requiring changes to existing ML code. Unfortunately, there is no corresponding way to eta-expand the functor application `MakeSet(C)` in the example from Figure 7.5. To statically ensure that the recursion in that example is safe, it appears that one *must* have access to the implementation of the `MakeSet` functor in order to instrument it with name abstractions and assign it a more precise non-strict interface.

This example also illustrates why it is useful to be able to instantiate a name abstraction with a *support* instead of a single name. In particular, suppose that `f`'s type were $\mathrm{D} \xrightarrow{\mathcal{S}} \mathrm{C}$ for some non-singleton support $\mathcal{S}$. The definition of `g` would become `map` $\mathcal{S}$ `f`, which is only possible given the ability to instantiate `map` with a support.

Finally, note that while my system does not contain any notion of subtyping, it is important to be able to coerce a non-strict function into an ordinary (potentially strict) arrow type. The coercion from $\forall\mathcal{X}.\mathsf{box}_{\mathcal{X}}(\mathrm{D}) \to \mathrm{C}$ to $\mathrm{D} \to \mathrm{C}[\emptyset/\mathcal{X}]$ is easily encodable within the language as $\lambda f.\lambda x.f(\emptyset)(\mathsf{box}(x))$.

[4] For simplicity, I am ignoring here that the result signature `SET` really depends on the type components of the functor argument.

7.2.4 Basic Declarative Properties

Here I give some basic declarative properties of the safe recursion language, for which the proofs are by straightforward induction. Note that, in Part 3 of Weakening and Part 1 of Substitution, the invariant $\mathcal{T} \subseteq \mathcal{S}$ is maintained inductively because the supports of the premises in every typing rule are supersets of the support of the conclusion.

Proposition 7.2.1 (Type Equivalence is an Equivalence Relation)
Type equivalence modulo $\mathcal{S}$ is an equivalence relation on well-formed types.

Proposition 7.2.2 (Weakening)
Suppose Γ is a prefix of Γ', and $\mathcal{S} \subseteq \mathcal{S}'$.

1. If $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]$, then $\Gamma' \vdash e : \mathrm{C}\ [\mathcal{S}']$.
2. If $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]$, then $\Gamma' \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}']$.
3. If $\Gamma, x : \mathrm{C}_1, \Gamma' \vdash e : \mathrm{C}\ [\mathcal{S}]$ and $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{T}]$ and $\mathcal{T} \subseteq \mathcal{S}$, then $\Gamma, x : \mathrm{C}_2, \Gamma' \vdash e : \mathrm{C}\ [\mathcal{S}]$.

Proposition 7.2.3 (Substitution)
1. If $\Gamma, x : \mathrm{C}, \Gamma' \vdash e : \mathrm{D}\ [\mathcal{S}]$ and $\Gamma \vdash v : \mathrm{C}\ [\mathcal{T}]$ and $\mathcal{T} \subseteq \mathcal{S}$, then $\Gamma, \Gamma' \vdash e[v/x] : \mathrm{D}\ [\mathcal{S}]$.
2. If $\Gamma, x : \mathrm{C}, \Gamma' \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]$, then $\Gamma, \Gamma' \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]$.
3. If $\Gamma, \mathcal{X}, \Gamma' \vdash e : \mathrm{C}\ [\mathcal{S}]$ and $\mathcal{T} \subseteq \mathrm{dom}(\Gamma)$, then $\Gamma, \Gamma'[\mathcal{T}/\mathcal{X}] \vdash e[\mathcal{T}/\mathcal{X}] : \mathrm{C}[\mathcal{T}/\mathcal{X}]\ [\mathcal{S}[\mathcal{T}/\mathcal{X}]]$.
4. If $\Gamma, \mathcal{X}, \Gamma' \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]$ and $\mathcal{T} \subseteq \mathrm{dom}(\Gamma)$, then $\Gamma, \Gamma'[\mathcal{T}/\mathcal{X}] \vdash \mathrm{C}_1[\mathcal{T}/\mathcal{X}] \equiv \mathrm{C}_2[\mathcal{T}/\mathcal{X}]\ [\mathcal{S}[\mathcal{T}/\mathcal{X}]]$.

7.2.5 Decidability of Typechecking

Figure 7.6 shows a straightforward typechecking algorithm for the safe recursion language. The algorithm takes as input a context Γ, a term e and a support $\mathcal{S}$, and synthesizes the unique type of e under the given support. As with the typing judgments, I make implicit assumptions about the well-formedness of the algorithmic judgments defined in Figure 7.6, *e.g.,* that all free names to the right of the turnstile are bound in the context.

The algorithm relies heavily on the fact that terms are explicitly-typed, and in particular that λ-abstractions specify the support of their bodies. The problem of inferring types for implicitly-typed terms is discussed in Section 7.6.1.

Decidability of the explicitly-typed system follows from the fact that the synthesis algorithm is syntax-directed, sound and complete. To prove soundness of the synthesis algorithm, we need the following technical lemma. Alternatively, we could modify the second premise of the typing rule for recursive terms (Rule 12) to be $\Gamma \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{S} \cup \mathcal{X}]$, in order to match the second premise of the corresponding synthesis rule. The lemma just shows that that modified version of the typing rule is already admissible.

Lemma 7.2.4 (Division of Support)
If $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ and $\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]$, then there exists a type C such that $\Gamma \vdash \mathrm{C} \equiv \mathrm{C}_1\ [\mathcal{S}_1]$ and $\Gamma \vdash \mathrm{C} \equiv \mathrm{C}_2\ [\mathcal{S}_2]$.

Type checking: $\Gamma \vdash e \Leftarrow \mathrm{C}\ [\mathcal{S}]$

$$\frac{\Gamma \vdash e \Rightarrow \mathrm{D}\ [\mathcal{S}] \quad \Gamma \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash e \Leftarrow \mathrm{C}\ [\mathcal{S}]}$$

Type synthesis: $\Gamma \vdash e \Rightarrow \mathrm{C}\ [\mathcal{S}]$

$$\frac{x : \mathrm{C} \in \Gamma}{\Gamma \vdash x \Rightarrow \mathrm{C}\ [\mathcal{S}]} \qquad \frac{}{\Gamma \vdash \langle\rangle \Rightarrow \mathsf{unit}\ [\mathcal{S}]} \qquad \frac{\Gamma \vdash v_1 \Rightarrow \mathrm{C}_1\ [\mathcal{S}] \quad \Gamma \vdash v_2 \Rightarrow \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \langle v_1, v_2\rangle \Rightarrow \mathrm{C}_1 \times \mathrm{C}_2\ [\mathcal{S}]} \qquad \frac{\Gamma \vdash v \Rightarrow \mathrm{C}_1 \times \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \pi_i(v) \Rightarrow \mathrm{C}_i\ [\mathcal{S}]}$$

$$\frac{\Gamma, x : \mathrm{D} \vdash e \Rightarrow \mathrm{C}\ [\mathcal{S} \cup \mathcal{T}]}{\Gamma \vdash \lambda^{\mathcal{T}} x : \mathrm{D}.\, e \Rightarrow \mathrm{D} \xrightarrow{\mathcal{T}} \mathrm{C}\ [\mathcal{S}]} \qquad \frac{\Gamma \vdash v_1 \Rightarrow \mathrm{D} \xrightarrow{\mathcal{T}} \mathrm{C}\ [\mathcal{S}] \quad \Gamma \vdash v_2 \Leftarrow \mathrm{D}\ [\mathcal{S}] \quad \mathcal{T} \subseteq \mathcal{S}}{\Gamma \vdash v_1(v_2) \Rightarrow \mathrm{C}\ [\mathcal{S}]}$$

$$\frac{\Gamma, \mathcal{X} \vdash e \Rightarrow \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \lambda\mathcal{X}.e \Rightarrow \forall\mathcal{X}.\mathrm{C}\ [\mathcal{S}]} \qquad \frac{\Gamma \vdash v \Rightarrow \forall\mathcal{X}.\mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash v(\mathcal{T}) \Rightarrow \mathrm{C}[\mathcal{T}/\mathcal{X}]\ [\mathcal{S}]}$$

$$\frac{\Gamma \vdash v \Rightarrow \mathrm{C}\ [\mathcal{S} \cup \mathcal{T}]}{\Gamma \vdash \mathsf{box}_{\mathcal{T}}(v) \Rightarrow \mathsf{box}_{\mathcal{T}}(\mathrm{C})\ [\mathcal{S}]} \qquad \frac{\Gamma \vdash v \Rightarrow \mathsf{box}_{\mathcal{T}}(\mathrm{C})\ [\mathcal{S}] \quad \mathcal{T} \subseteq \mathcal{S}}{\Gamma \vdash \mathsf{unbox}(v) \Rightarrow \mathrm{C}\ [\mathcal{S}]}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \mathrm{D}\ [\mathcal{S}] \quad \Gamma, x : \mathrm{D} \vdash e_2 \Rightarrow \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Rightarrow \mathrm{C}\ [\mathcal{S}]}$$

$$\frac{\Gamma, \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}) \vdash e \Rightarrow \mathrm{D}\ [\mathcal{S}] \quad \Gamma, \mathcal{X} \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{S} \cup \mathcal{X}]}{\Gamma \vdash \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e) \Rightarrow \mathrm{C}\ [\mathcal{S}]}$$

Figure 7.6: Typechecking Algorithm for Safe Recursion Language

Proof: By induction on derivations. The only interesting cases are when C_1 and C_2 carry a support, *e.g.,* when C_i is of the form $\mathsf{box}_{\mathcal{T}_i}(\mathrm{D}_i)$. In this case, define $\mathcal{T} := (\mathcal{T}_1 - \mathcal{S}_1) \cup (\mathcal{T}_2 - \mathcal{S}_2)$.

First we need to show that $\mathcal{T} \cup \mathcal{S}_i = \mathcal{T}_i \cup \mathcal{S}_i$. We will show this for $i = 1$, the proof for $i = 2$ is completely symmetric. We are given that $\mathcal{T}_2 \cup (\mathcal{S}_1 \cup \mathcal{S}_2) = \mathcal{T}_1 \cup (\mathcal{S}_1 \cup \mathcal{S}_2)$. Thus, $\mathcal{T}_2 \subseteq \mathcal{T}_1 \cup \mathcal{S}_1 \cup \mathcal{S}_2$. Subtracting $\mathcal{S}_2$ from both sides, $\mathcal{T}_2 - \mathcal{S}_2 \subseteq (\mathcal{T}_1 \cup \mathcal{S}_1 \cup \mathcal{S}_2) - \mathcal{S}_2 \subseteq \mathcal{T}_1 \cup \mathcal{S}_1$. Now, expanding out the definition of $\mathcal{T}$, we have that $\mathcal{T} \cup \mathcal{S}_1 = (\mathcal{T}_1 \cup \mathcal{S}_1) \cup (\mathcal{T}_2 - \mathcal{S}_2)$. Then since $\mathcal{T}_2 - \mathcal{S}_2 \subseteq \mathcal{T}_1 \cup \mathcal{S}_1$, the right-hand side is equal to $\mathcal{T}_1 \cup \mathcal{S}_1$.

We are given also that $\Gamma \vdash \mathrm{D}_1 \equiv \mathrm{D}_2\ [\mathcal{S} \cup \mathcal{T}_1]$. Expanding out $\mathcal{T}$ it is clear that $\mathcal{S} \cup \mathcal{T} = \mathcal{S} \cup \mathcal{T}_1 \cup \mathcal{T}_2 = \mathcal{S} \cup \mathcal{T}_1 = \mathcal{S} \cup \mathcal{T}_2$. Now define $\mathcal{S}_i' := \mathcal{S}_i \cup \mathcal{T}$. Clearly, $\mathcal{S}_1' \cup \mathcal{S}_2' = \mathcal{S} \cup \mathcal{T}$. By induction, there exists D such that $\Gamma \vdash \mathrm{D} \equiv \mathrm{D}_i\ [\mathcal{S}_i']$. Define $\mathrm{C} := \mathsf{box}_{\mathcal{T}}(\mathrm{D})$. By Rule 18, $\Gamma \vdash \mathrm{C} \equiv \mathsf{box}_{\mathcal{T}_i}(\mathrm{D}_i)\ [\mathcal{S}_i]$. ■

Theorem 7.2.5 (Soundness of Algorithm)
If $\Gamma \vdash e \Rightarrow \mathrm{C}\ [\mathcal{S}]$ or $\Gamma \vdash e \Leftarrow \mathrm{C}\ [\mathcal{S}]$, then $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]$.

Proof: By straightforward induction on the algorithm. The only interesting case is the synthesis rule for recursive terms. For that case, we know by induction that $\Gamma, \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}) \vdash e : \mathrm{D}\ [\mathcal{S}]$. By Lemma 7.2.4, since $\Gamma, \mathcal{X} \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{S} \cup \mathcal{X}]$, we know that there exists D' such that $\Gamma, \mathcal{X} \vdash \mathrm{D} \equiv \mathrm{D}'\ [\mathcal{S}]$ and $\Gamma, \mathcal{X} \vdash \mathrm{D}' \equiv \mathrm{C}\ [\mathcal{X}]$. By Rule 13, $\Gamma, \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}) \vdash e : \mathrm{D}'\ [\mathcal{S}]$, so the desired result follows by Rule 12. ■

Machine States	Ω ::=	$(\omega; \mathcal{C}; e)$
Continuations	$\mathcal{C}$ ::=	$\bullet \mid \mathcal{C} \circ \mathcal{F}$
Continuation Frames	$\mathcal{F}$ ::=	$\mathsf{let}\ x = \bullet\ \mathsf{in}\ e \mid \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, \bullet)$

$\boxed{\textbf{Small-step semantics:}\ \ \Omega \mapsto \Omega'}$

$$\frac{}{(\omega; \mathcal{C}; \pi_i\langle v_1, v_2\rangle) \mapsto (\omega; \mathcal{C}; v_i)}\ (19) \qquad \frac{}{(\omega; \mathcal{C}; (\lambda^{\mathcal{T}} x : \mathrm{C}.\, e)(v)) \mapsto (\omega; \mathcal{C}; e[v/x])}\ (20)$$

$$\frac{}{(\omega; \mathcal{C}; (\lambda\mathcal{X}.e)(\mathcal{T})) \mapsto (\omega; \mathcal{C}; e[\mathcal{T}/\mathcal{X}])}\ (21) \qquad \frac{x \notin \mathrm{dom}(\omega)}{(\omega; \mathcal{C}; \mathsf{box}_{\mathcal{T}}(v)) \mapsto (\omega[x \mapsto v]; \mathcal{C}; x)}\ (22)$$

$$\frac{x \in \mathrm{dom}(\omega) \quad \omega(x) = v}{(\omega; \mathcal{C}; \mathsf{unbox}(x)) \mapsto (\omega; \mathcal{C}; v)}\ (23) \qquad \frac{}{(\omega; \mathcal{C}; \mathsf{let}\ x = e'\ \mathsf{in}\ e) \mapsto (\omega; \mathcal{C} \circ \mathsf{let}\ x = \bullet\ \mathsf{in}\ e; e')}\ (24)$$

$$\frac{}{(\omega; \mathcal{C} \circ \mathsf{let}\ x = \bullet\ \mathsf{in}\ e; v) \mapsto (\omega; \mathcal{C}; e[v/x])}\ (25)$$

$$\frac{\mathcal{X}, x \notin \mathrm{dom}(\omega)}{(\omega; \mathcal{C}; \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e)) \mapsto (\omega[\mathcal{X} \mapsto x][x \mapsto ?]; \mathcal{C} \circ \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, \bullet); e)}\ (26)$$

$$\frac{x \in \mathrm{dom}(\omega)}{(\omega; \mathcal{C} \circ \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, \bullet); v) \mapsto (\omega[x := v]; \mathcal{C}; v)}\ (27)$$

Figure 7.7: Dynamic Semantics for Safe Recursion Language

Theorem 7.2.6 (Completeness of Algorithm)
If $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]$, then $\Gamma \vdash e \Leftarrow \mathrm{C}\ [\mathcal{S}]$.

Proof: By straightforward induction on derivations. Again, the only interesting case is the typing rule for recursive terms. By induction, $\Gamma, \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}) \vdash e \Rightarrow \mathrm{D}'\ [\mathcal{S}]$ and $\Gamma \vdash \mathrm{D}' \equiv \mathrm{D}\ [\mathcal{S}]$. Since the second premise of the typing rule tells us that $\Gamma \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{X}]$, we have by Weakening that $\Gamma \vdash \mathrm{D}' \equiv \mathrm{C}\ [\mathcal{S} \cup \mathcal{X}]$. Thus, by definition of synthesis, $\Gamma \vdash \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e) \Rightarrow \mathrm{C}\ [\mathcal{S}]$. (It is critical here that the second premise of the synthesis rule for recursive terms use the modulus $\mathcal{S} \cup \mathcal{X}$ instead of just $\mathcal{X}$.) ■

7.2.6 Dynamic Semantics and Type Safety

I formalize the dynamic semantics for safe recursive terms in Figure 7.7, using an abstract machine semantics very similar to the one developed in Section 6.2. The main differences are as follows.

First, there is no need for the machine state Error in the present semantics. Since the type system ensures that a recursive variable will never be unboxed before it has been backpatched, the Error state will never arise. Formally speaking, there is only one rule in the dynamic semantics for $\mathsf{unbox}(v)$ (Rule 23), and that rule only applies if the contents of location v are defined. If the contents of v are not defined, the machine will get *stuck*, and the type safety theorem guarantees that this cannot happen.

Second, in order to connect names $\mathcal{X}$ to their associated locations, machine stores ω now contain mappings for both locations (represented as variables) and names. As in Section 6.2, variables are mapped to either values (v) or junk ($\mathbf{?}$). Names, in turn, are mapped to variables. I will write $\omega[\mathcal{X} \mapsto x]$ to indicate the extension of ω with the mapping of $\mathcal{X}$ to x, and $\omega(\mathcal{X})$ to indicate the variable to which $\mathcal{X}$ is mapped in ω. Thus, in Rule 26 for evaluating recursive terms, x is added to the store with a binding to junk, and $\mathcal{X}$ is added to the store with a binding to x.[5]

Otherwise, the dynamic semantics of Figure 7.7 should be self-explanatory.

Before proving type safety, I would like to stress an important point about the practical implementation of this semantics. Formally, my dynamic semantics treats values of type $\mathsf{box}_{\mathcal{S}}(\mathrm{C})$ as memory locations that will eventually contain values of type C. It is quite likely, though, that values of type C (*i.e.,* the kinds of values one wants to define recursively) have a naturally boxed representation. For instance, in the case of recursive modules, C will typically be a record type, and a module value of type C will be represented as a pointer to a record stored on the heap.

Only one level of pointer indirection is needed to implement backpatching. Thus, a direct implementation of my semantics that represents all values of type $\mathsf{box}_{\mathcal{S}}(\mathrm{C})$ as pointers to values of type C will introduce an unnecessary level of indirection when values of type C are already pointers. My semantics, however, does not require one to employ such a naïve representation. Indeed, for types C with naturally boxed representations, a realistic implementation should represent values of type $\mathsf{box}_{\mathcal{S}}(\mathrm{C})$ the same as values of type C and should compile the unbox'ing of such values as a no-op. (See Hirschowitz *et al.* [36] for an example of such a compilation strategy.) At the level of the type system, though, there is still an important semantic distinction to be made between C and $\mathsf{box}_{\mathcal{S}}(\mathrm{C})$ that transcends such implementation details.

To prove type safety, I will again follow the approach of Section 6.2 quite closely.

Definition 7.2.7 (Run-Time Contexts)
A context Γ is *run-time* if the only entries in Γ have the form $\mathcal{X}$ or $x : \mathsf{box}_{\mathcal{T}}(\mathrm{C})$.

Definition 7.2.8 (Machine Store Well-formedness)
A machine store ω is *well-formed*, denoted $\Gamma \vdash \omega\ [\mathcal{S}]$, if:

1. Γ is run-time and $\mathrm{dom}(\omega) = \mathrm{dom}(\Gamma)$

2. $\forall \mathcal{X} \in \Gamma.\ \omega(\mathcal{X}) = x$ if and only if $\Gamma = \Gamma', \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}), \Gamma''$ for some C

3. $\forall x : \mathsf{box}_{\mathcal{T}}(\mathrm{C}) \in \Gamma.$ either $(\omega(x) = v$, where $\Gamma \vdash v : \mathrm{C}\ [\mathcal{S} \cup \mathcal{T}])$
or $(\omega(x) = \mathbf{?}$, and $\exists \mathcal{X} \notin \mathcal{S}.\ \omega(\mathcal{X}) = x)$

Like the machine store well-formedness judgment defined in Section 6.2, the judgment defined in Definition 7.2.8 uses a typing context Γ to describe the store ω. As stores now bind names as well as variables, the second condition of the judgment asserts that the location x to which a name $\mathcal{X}$ is mapped in the store is the same as the variable to which it is associated (by juxtaposition) in the typing context. Incidentally, this condition also guarantees that the store maps every name to a different recursive variable.

The store well-formedness judgment differs from the earlier one in that it also includes a support $\mathcal{S}$. The idea is that a name $\mathcal{X}$ will only appear in the support $\mathcal{S}$ if the recursive variable associated with that name has been backpatched. Correspondingly, the third condition of the judgment asserts that for every location x in the store, the only way that x may be bound to junk is if its associated name is not in the support $\mathcal{S}$.

[5] In reality, names can be erased during code generation, so the store does not actually have to create bindings for them, but it is useful for semantic purposes to view the store as doing so.

Well-formed continuations: $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$

$$\frac{}{\Gamma \vdash \bullet : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]}\ (28) \qquad \frac{\Gamma \vdash \mathcal{F} : \mathrm{C} \Rightarrow \mathrm{D}\ [\mathcal{S}] \quad \Gamma \vdash \mathcal{C} : \mathrm{D}\ \mathsf{cont}\ [\mathcal{S}]}{\Gamma \vdash \mathcal{C} \circ \mathcal{F} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]}\ (29)$$

$$\frac{\Gamma \vdash \mathcal{C} : \mathrm{D}\ \mathsf{cont}\ [\mathcal{S}] \quad \Gamma \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]}\ (30)$$

Well-formed continuation frames: $\Gamma \vdash \mathcal{F} : \mathrm{C}_1 \Rightarrow \mathrm{C}_2\ [\mathcal{S}]$

$$\frac{\Gamma, x{:}\mathrm{C}_1 \vdash e : \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \mathsf{let}\ x = \bullet\ \mathsf{in}\ e : \mathrm{C}_1 \Rightarrow \mathrm{C}_2\ [\mathcal{S}]}\ (31) \qquad \frac{\Gamma = \Gamma', \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}), \Gamma'' \quad \Gamma \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{X}]}{\Gamma \vdash \mathsf{saferec}(\mathcal{X} \rhd x : \mathrm{C}.\ \bullet) : \mathrm{D} \Rightarrow \mathrm{C}\ [\mathcal{S}]}\ (32)$$

Figure 7.8: Well-Formed Continuations for Safe Recursion Language

Figure 7.8 defines well-formedness of continuations and continuation frames. The only rule that is slightly unusual is Rule 32 for recursive frames $\mathsf{saferec}(\mathcal{X} \rhd x : \mathrm{C}.\ \bullet)$. This frame is not a binder for $\mathcal{X}$ or x; rather, Rule 32 requires that $\mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C})$ appear in the context. This is a safe assumption since $\mathsf{saferec}(\mathcal{X} \rhd x : \mathrm{C}.\ \bullet)$ only gets pushed on the stack after bindings for $\mathcal{X}$ and x have been added to the store.

We can now define a notion of well-formedness for a machine state, and prove type safety. I only show the proof for the interesting cases, *i.e.,* recursive terms and unboxing. For full proofs, I refer the reader to Dreyer *et al.* [13].

Definition 7.2.9 (Machine State Well-formedness)
A machine state Ω is *well-formed*, denoted $\Gamma \vdash \Omega\ [\mathcal{S}]$, if $\Omega = (\omega; \mathcal{C}; e)$, where:

1. $\Gamma \vdash \omega\ [\mathcal{S}]$
2. $\exists \mathrm{C}.\ \Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$ and $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]$

Proposition 7.2.10 (Weakening for Continuations)
Suppose Γ is a prefix of Γ', and $\mathcal{S} \subseteq \mathcal{S}'$.

1. If $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$, then $\Gamma' \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}']$.
2. If $\Gamma \vdash \mathcal{F} : \mathrm{C}_1 \Rightarrow \mathrm{C}_2\ [\mathcal{S}]$, then $\Gamma' \vdash \mathcal{F} : \mathrm{C}_1 \Rightarrow \mathrm{C}_2\ [\mathcal{S}']$.

Theorem 7.2.11 (Preservation)
If $\Gamma \vdash \Omega\ [\mathcal{S}]$ and $\Omega \mapsto \Omega'$, then $\exists\, \Gamma', \mathcal{S}'.\ \Gamma' \vdash \Omega'\ [\mathcal{S}']$.

Proof: By cases on the second premise.

- Case: Rule 23.
  1. By assumption, $\Gamma \vdash \omega\ [\mathcal{S}]$, $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$, $\Gamma \vdash \mathsf{unbox}(x) : \mathrm{C}\ [\mathcal{S}]$, and $\omega(x) = v$.
  2. By inversion on synthesis, $x : \mathsf{box}_{\mathcal{T}}(\mathrm{D}) \in \Gamma$, where $\mathcal{T} \subseteq \mathcal{S}$, and $\Gamma \vdash \mathrm{C} \equiv \mathrm{D}\ [\mathcal{S}]$.
  3. By condition 3 of $\Gamma \vdash \omega\ [\mathcal{S}]$, since $\omega(x) = v$, we have $\Gamma \vdash v : \mathrm{C}\ [\mathcal{S}]$.

- Case: Rule 26.

  1. By assumption, $\Gamma \vdash \omega\ [\mathcal{S}]$, $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$, and $\Gamma \vdash \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e) : \mathrm{C}\ [\mathcal{S}]$.
  2. Let $\Gamma' = \Gamma, \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C})$.
  3. By inversion on synthesis, there exists D such that $\Gamma' \vdash e : \mathrm{D}\ [\mathcal{S}]$ and $\Gamma' \vdash \mathrm{D} \equiv \mathrm{C}\ [X]$.
  4. By Weakening, $\Gamma' \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$.
  5. By Rule 32, $\Gamma' \vdash \mathcal{C} \circ \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, \bullet) : \mathrm{D}\ \mathsf{cont}\ [\mathcal{S}]$.
  6. By Weakening and since $\mathcal{X} \notin \mathcal{S}$, we have $\Gamma' \vdash \omega[\mathcal{X} \mapsto x][x \mapsto \mathbf{?}]\ [\mathcal{S}]$.

- Case: Rule 27.

  1. By assumption, $\Gamma \vdash \omega\ [\mathcal{S}]$, $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$, $\Gamma \vdash v : \mathrm{D}\ [\mathcal{S}]$, $\Gamma \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{X}]$ and $\Gamma = \Gamma', \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}), \Gamma''$.
  2. By Weakening and Rule 13, $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S} \cup \mathcal{X}]$ and $\Gamma \vdash v : \mathrm{C}\ [\mathcal{S} \cup \mathcal{X}]$.
  3. Thus, also by Weakening, $\Gamma \vdash \omega[x := v]\ [\mathcal{S} \cup \mathcal{X}]$.

■

Lemma 7.2.12 (Canonical Forms)
Suppose that Γ is run-time and $\Gamma \vdash v : \mathrm{C}\ [\mathcal{S}]$.

1. If $\mathrm{C} = \mathsf{unit}$, then v is of the form $\langle\rangle$.
2. If $\mathrm{C} = \mathrm{C}_1 \times \mathrm{C}_2$, then v is of the form $\langle v_1, v_2 \rangle$.
3. If $\mathrm{C} = \mathrm{C}_1 \xrightarrow{\mathcal{T}} \mathrm{C}_2$, then v is of the form $\lambda^{\mathcal{S}'} x : \mathrm{D}.\, e$.
4. If $\mathrm{C} = \forall \mathcal{X}.\mathrm{D}$, then v is of the form $\lambda \mathcal{X}.e$.
5. Otherwise, v is a variable x.

Definition 7.2.13 (Terminal States)
A machine state Ω is *terminal* if it has the form $(\omega; \bullet; v)$.

Definition 7.2.14 (Stuck States)
A machine state Ω is *stuck* if it is non-terminal and there is no state Ω' such that $\Omega \mapsto \Omega'$.

Theorem 7.2.15 (Progress)
If $\Gamma \vdash \Omega\ [\mathcal{S}]$, then Ω is not stuck.

Proof: Assume $\Omega = (\omega; \mathcal{C}; e)$. By assumption, $\Gamma \vdash \omega\ [\mathcal{S}]$. The proof is by cases on $\mathcal{C}$ and e.

- Case: $e = \mathsf{unbox}(v)$.

  1. By assumption, $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$ and $\Gamma \vdash \mathsf{unbox}(v) : \mathrm{C}\ [\mathcal{S}]$.
  2. By inversion on synthesis and Canonical Forms, v has the form x,
     where $x : \mathsf{box}_{\mathcal{T}}(\mathrm{D}) \in \Gamma$ and $\mathcal{T} \subseteq \mathcal{S}$ and $\Gamma \vdash \mathrm{D} \equiv \mathrm{C}\ [\mathcal{S}]$.
  3. By condition 1 of $\Gamma \vdash \omega\ [\mathcal{S}]$, $x \in \mathrm{dom}(\omega)$.
  4. By condition 2 of $\Gamma \vdash \omega\ [\mathcal{S}]$, if there exists $\mathcal{X}$ such that $\omega(\mathcal{X}) = x$, then $\mathcal{X} \in \mathcal{S}$.

$$\begin{array}{llcl} \text{Types} & \mathrm{C} & ::= & \cdots \mid \mathsf{ref}(\mathrm{C}) \mid \mathsf{cont}(\mathrm{C}) \\ \text{Terms} & e & ::= & \cdots \mid \mathsf{ref}(v) \mid \mathsf{get}(v) \mid \mathsf{set}(v_1, v_2) \mid \mathsf{callcc}_{\mathrm{C}}(x.\, e) \mid \mathsf{throw}_{\mathrm{C}}(v_1, v_2) \end{array}$$

$$\frac{\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \mathsf{ref}(\mathrm{C}_1) \equiv \mathsf{ref}(\mathrm{C}_2)\ [\mathcal{S}]}\ (33) \qquad \frac{\Gamma \vdash v : \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \mathsf{ref}(v) : \mathsf{ref}(\mathrm{C})\ [\mathcal{S}]}\ (34)$$

$$\frac{\Gamma \vdash v : \mathsf{ref}(\mathrm{C})\ [\mathcal{S}]}{\Gamma \vdash \mathsf{get}(v) : \mathrm{C}\ [\mathcal{S}]}\ (35) \qquad \frac{\Gamma \vdash v_1 : \mathsf{ref}(\mathrm{C})\ [\mathcal{S}] \quad \Gamma \vdash v_2 : \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \mathsf{set}(v_1, v_2) : 1\ [\mathcal{S}]}\ (36)$$

$$\frac{\Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{S}]}{\Gamma \vdash \mathsf{cont}(\mathrm{C}_1) \equiv \mathsf{cont}(\mathrm{C}_2)\ [\mathcal{S}]}\ (37) \qquad \frac{\Gamma, x : \mathsf{cont}(\mathrm{C}) \vdash e : \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \mathsf{callcc}_{\mathrm{C}}(x.\, e) : \mathrm{C}\ [\mathcal{S}]}\ (38)$$

$$\frac{\Gamma \vdash v_1 : \mathsf{cont}(\mathrm{D})\ [\mathcal{S}] \quad \Gamma \vdash v_2 : \mathrm{D}\ [\mathcal{S}]}{\Gamma \vdash \mathsf{throw}_{\mathrm{C}}(v_1, v_2) : \mathrm{C}\ [\mathcal{S}]}\ (39)$$

Figure 7.9: Static Semantics Extensions for References and Continuations

5. Then, by condition 3 of $\Gamma \vdash \omega\ [\mathcal{S}]$, it cannot be the case that $\omega(x) = \mathbf{?}$.
6. Thus, there exists v' such that $\omega(x) = v'$, and Ω makes a step by Rule 23.

- Case: $e = \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e')$. Ω makes a step by Rule 26.
- Case: $e = v$, and $\mathcal{C} = \mathcal{C}' \circ \mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, \bullet)$.
  1. By assumption, since $\mathcal{C}$ is well-formed, we have $\Gamma = \Gamma', \mathcal{X}, x : \mathsf{box}_{\mathcal{X}}(\mathrm{C}), \Gamma''$.
  2. By condition 1 of $\Gamma \vdash \omega\ [\mathcal{S}]$, $x \in \mathrm{dom}(\omega)$, so Ω makes a step by Rule 27.

■

Corollary 7.2.16 (Type Safety)
If $\emptyset \vdash e : \mathrm{C}\ [\emptyset]$, then the evaluation of $(\epsilon; \bullet; e)$ will not get stuck.

7.3 Adding Computational Effects

Since I have modeled the semantics of backpatching in terms of a mutable store, it is easy to incorporate some actual computational effects into the language. Figure 7.9 extends the syntax and static semantics of the safe recursion language with mutable state and continuations. The primitives for the former are ref, get and set, and the primitives for the latter are callcc and throw, all with the standard typing rules. Ref cells and continuations are allocated in the store, so the values of types $\mathsf{ref}(\mathrm{C})$ and $\mathsf{cont}(\mathrm{C})$ are variables representing locations in the store.

If we think of a continuation as a kind of function with no return type, it may seem surprising that the typing rules for continuations are oblivious to names. Moreover, while the arrow type $\mathrm{C}_1 \xrightarrow{\mathcal{S}} \mathrm{C}_2$ specifies the support $\mathcal{S}$ required to call a function of that type, the continuation type $\mathsf{cont}(\mathrm{C})$ does not specify a support, and no support is required in order to throw to a continuation. (One can view $\mathsf{cont}(\mathrm{C})$ as always specifying empty support.)

To see why a support is unnecessary, consider what the machine state looks like when we are about to evaluate a callcc: it has the form $\Omega = (\omega; \mathcal{C}; \mathsf{callcc}_{\mathrm{C}}(x.\, e))$. Assuming that Ω is well-formed ($\Gamma \vdash \Omega\ [\mathcal{S}]$), we know that $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$ and $\Gamma \vdash \mathsf{callcc}_{\mathrm{C}}(x.\, e) : \mathrm{C}\ [\mathcal{S}]$. The current continuation

$$\frac{x \notin \mathrm{dom}(\omega)}{(\omega; \mathcal{C}; \mathsf{ref}(v)) \mapsto (\omega[x \mapsto v]; \mathcal{C}; x)}\ (40)$$

$$\frac{x \in \mathrm{dom}(\omega) \quad \omega(x) = v}{(\omega; \mathcal{C}; \mathsf{get}(x)) \mapsto (\omega; \mathcal{C}; v)}\ (41) \qquad \frac{x \in \mathrm{dom}(\omega)}{(\omega; \mathcal{C}; \mathsf{set}(x, v)) \mapsto (\omega[x := v]; \mathcal{C}; \langle\rangle)}\ (42)$$

$$\frac{x \notin \mathrm{dom}(\omega)}{(\omega; \mathcal{C}; \mathsf{callcc}_{\mathrm{C}}(x.\, e)) \mapsto (\omega[x \mapsto \mathcal{C}]; \mathcal{C}; e)}\ (43) \qquad \frac{x \in \mathrm{dom}(\omega) \quad \omega(x) = \mathcal{C}_x}{(\omega; \mathcal{C}; \mathsf{throw}_{\mathrm{C}}(x, v)) \mapsto (\omega; \mathcal{C}_x; v)}\ (44)$$

Figure 7.10: Dynamic Semantics Extensions for References and Continuations

is $\mathcal{C}$ and that is what callcc will bind to x before evaluating e. Then what is the "type" of $\mathcal{C}$? Although explicit continuations are not part of our language, we can nonetheless think of $\mathcal{C}$ as a function with argument type C that, when applied, may dereference any of the recursive variables associated with the names in $\mathcal{S}$. Thus, the most appropriate arrow-like type for $\mathcal{C}$ would be $\mathrm{C} \xrightarrow{\mathcal{S}} \mathrm{D}$ for some return type D. Under support $\mathcal{S}$, though, this arrow type is equivalent to $\mathrm{C} \xrightarrow{\emptyset} \mathrm{D}$, or in other words $\mathsf{cont}(\mathrm{C})$.

Figure 7.10 gives the extensions to the dynamic semantics for mutable state and continuations. We extend stores ω to contain mappings from locations x to continuations $\mathcal{C}$. The rules for mutable state are completely straightforward. The rules for continuations are also fairly straightforward, since the machine state already makes the current continuation explicit. Proving type safety for these extensions requires only a simple, orthogonal extension of the proof framework from Section 7.2.6. The definition of run-time contexts is extended to include variables of type $\mathsf{ref}(\mathrm{C})$ and $\mathsf{cont}(\mathrm{C})$, and the definition of store well-formedness is extended as follows:

Definition 7.3.1 (Run-Time Contexts)
A context Γ is *run-time* if the only bindings in Γ have the form $\mathcal{X}$, $x : \mathsf{box}_{\mathcal{T}}(\mathrm{C})$, $x : \mathsf{ref}(\mathrm{C})$ or $x : \mathsf{cont}(\mathrm{C})$.

Definition 7.3.2 (Machine Store Well-formedness)
A store ω is *well-formed*, denoted $\Gamma \vdash \omega\ [\mathcal{S}]$, if $\Gamma \vdash \omega\ [\mathcal{S}]$ according to Definition 7.2.8 and also:

1. $\forall x : \mathsf{ref}(\mathrm{C}) \in \Gamma.\ \exists v.\ \omega(x) = v$ and $\Gamma \vdash v : \mathrm{C}\ [\mathcal{S}]$
2. $\forall x : \mathsf{cont}(\mathrm{C}) \in \Gamma.\ \exists \mathcal{C}.\ \omega(x) = \mathcal{C}$ and $\Gamma \vdash \mathcal{C} : \mathrm{C}\ \mathsf{cont}\ [\mathcal{S}]$

7.4 Encoding Unrestricted Recursion

Despite all the efforts of the type system, there will always be recursive terms $\mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e)$ for which we cannot statically determine that e can be evaluated without dereferencing x. For such cases it is important for the programmer to have the fallback approach of using the unrestricted recursive term construct $\mathsf{rec}(x : \mathrm{C}.\, e)$ defined in Chapter 6, with the understanding that dereferences of x will be saddled with a corresponding run-time cost in order to guarantee type safety.

The point of this section is to illustrate that the unrestricted $\mathsf{rec}(x : \mathrm{C}.\, e)$ may be encoded in terms of $\mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e)$ if we extend the language with primitives for *memoized computations*. The syntax and static semantics of this extension are given in Figure 7.11. First, we introduce a

$$\begin{array}{lrl} \text{Types} & \mathrm{C} ::= & \cdots \mid \mathsf{comp}_{\mathcal{S}}(\mathrm{C}) \\ \text{Terms} & e ::= & \cdots \mid \mathsf{delay}_{\mathcal{S}}(e) \mid \mathsf{force}(v) \end{array}$$

$$\frac{\mathcal{S} \cup \mathcal{S}_1 = \mathcal{T} = \mathcal{S} \cup \mathcal{S}_2 \quad \Gamma \vdash \mathrm{C}_1 \equiv \mathrm{C}_2\ [\mathcal{T}]}{\Gamma \vdash \mathsf{comp}_{\mathcal{S}_1}(\mathrm{C}_1) \equiv \mathsf{comp}_{\mathcal{S}_2}(\mathrm{C}_2)\ [\mathcal{S}]}\ (45)$$

$$\frac{\Gamma \vdash e : \mathrm{C}\ [\mathcal{S} \cup \mathcal{T}]}{\Gamma \vdash \mathsf{delay}(e) : \mathsf{comp}_{\mathcal{T}}(\mathrm{C})\ [\mathcal{S}]}\ (46)$$

$$\frac{\Gamma \vdash v : \mathsf{comp}_{\mathcal{T}}(\mathrm{C})\ [\mathcal{S}] \quad \mathcal{T} \subseteq \mathcal{S}}{\Gamma \vdash \mathsf{force}(v) : \mathrm{C}\ [\mathcal{S}]}\ (47)$$

Figure 7.11: Static Semantics Extensions for Memoized Computations

type $\mathsf{comp}_{\mathcal{S}}(\mathrm{C})$ of locations storing memoized computations. A value of this type is essentially a thunk of type $\mathsf{unit} \xrightarrow{\mathcal{S}} \mathrm{C}$ whose result is memoized after the first application.

The primitive $\mathsf{delay}_{\mathcal{S}}(e)$ creates a memoized location x in the store bound to the suspended term e. When x is forced (by $\mathsf{force}(x)$), the expression e stored at x is evaluated to a value v, and then x is backpatched with v. During the evaluation of e, the location x is bound to junk; if x is forced again during this stage, the machine raises an error. Thus, every force of x must check to see whether it is bound to an expression or junk. Despite the difference in operational behavior, the typing rules for memoized computations appear just as if $\mathsf{comp}_{\mathcal{S}}(\mathrm{C})$, $\mathsf{delay}_{\mathcal{S}}(e)$ and $\mathsf{force}(v)$ were shorthand for $\mathsf{unit} \xrightarrow{\mathcal{S}} \mathrm{C}$, $\lambda^{\mathcal{S}} x : \mathsf{unit}.\, e$ and $v\,\langle\rangle$, respectively. I use $\mathsf{comp}(\mathrm{C})$ and $\mathsf{delay}(e)$ as shorthand for $\mathsf{comp}_{\emptyset}(\mathrm{C})$ and $\mathsf{delay}_{\emptyset}(e)$, respectively.

We can now encode an unrestricted form of recursion. This construct has a typing rule similar to the one given in Section 6.2:

$$\frac{\Gamma, x : \mathsf{comp}(\mathrm{C}) \vdash e : \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash \mathsf{rec}(x : \mathrm{C}.\, e) : \mathrm{C}\ [\mathcal{S}]}$$

The recursive variable x is dereferenced by writing $\mathsf{force}(x)$ (similar to $\mathsf{fetch}(x)$). The encoding is as follows:

$$\mathsf{rec}(y : \mathrm{C}.\, e) \quad \stackrel{\text{def}}{=} \quad \mathsf{force}(\mathsf{saferec}(\mathcal{X} \triangleright x : \mathsf{comp}_{\emptyset}(\mathrm{C}).\, \mathsf{delay}_{\mathcal{X}}(\mathsf{let}\ y = \mathsf{unbox}(x)\ \mathsf{in}\ e)))$$

It is easiest to understand this encoding by stepping through it. First, a new *recursive* location x is created, bound to junk. Then, the delay creates a new *memoized* location z bound to the expression $\mathsf{let}\ y = \mathsf{unbox}(x)\ \mathsf{in}\ e$. Next, the $\mathsf{saferec}$ backpatches x with the value z and returns z. Finally, z is forced, resulting in the evaluation of $\mathsf{let}\ y = \mathsf{unbox}(x)\ \mathsf{in}\ e$, which steps to $e[z/y]$. Assuming this evaluates to a value v, the location z will then be backpatched with v. If z is dereferenced during the evaluation of $e[z/y]$ (by an invocation of $\mathsf{force}(y)$ in the original e), then a run-time error will be reported.

Essentially, one can view the $\mathsf{saferec}$ in this encoding as tying the recursive knot on the memoized computation, while the memoization resulting from the force is what actually performs the backpatching. Observe that if we were to give $\mathsf{comp}(\mathrm{C})$ a non-memoizing semantics, *i.e.,* to consider it synonymous with $\mathsf{unit} \to \mathrm{C}$, the above encoding would have precisely the fixed-point semantics

$$\begin{array}{lll} \text{Machine States} & \Omega ::= & \cdots \mid \mathsf{Error} \\ \text{Continuation Frames} & F ::= & \cdots \mid \mathsf{fill}\ x\ \mathsf{with}\ \bullet \end{array}$$

$$\frac{x \notin \mathrm{dom}(\omega)}{(\omega; \mathcal{C}; \mathsf{delay}_{\mathcal{S}}(e)) \mapsto (\omega[x \mapsto e]; \mathcal{C}; x)}\ (48)$$

$$\frac{x \in \mathrm{dom}(\omega) \quad \omega(x) = e}{(\omega; \mathcal{C}; \mathsf{force}(x)) \mapsto (\omega[x := \mathbf{?}]; \mathcal{C} \circ \mathsf{fill}\ x\ \mathsf{with}\ \bullet; e)}\ (49)$$

$$\frac{x \in \mathrm{dom}(\omega)}{(\omega; \mathcal{C} \circ \mathsf{fill}\ x\ \mathsf{with}\ \bullet; v) \mapsto (\omega[x := v]; \mathcal{C}; v)}\ (50)$$

$$\frac{x \in \mathrm{dom}(\omega) \quad \omega(x) = \mathbf{?}}{(\omega; \mathcal{C} \circ \mathsf{force}(\bullet); x) \mapsto \mathsf{Error}}\ (51)$$

$$\frac{\Gamma \vdash x : \mathsf{comp}(\mathrm{C})\ [\mathcal{S}]}{\Gamma \vdash \mathsf{fill}\ x\ \mathsf{with}\ \bullet : \mathrm{C} \Rightarrow \mathrm{C}\ [\mathcal{S}]}\ (52)$$

Figure 7.12: Dynamic Semantics Extensions for Memoized Computations

of recursion. Memoization ensures that the effects in e only happen once, at the first force of the recursive computation.

The dynamic semantics for this extension is given in Figure 7.12. To evaluate $\mathsf{delay}_{\mathcal{S}}(e)$, we create a new memoized location in the store and bind e to it (Rule 48). To evaluate $\mathsf{force}(x)$, we proceed to evaluate the term e to which x is bound, pushing on the continuation stack a memoization frame ($\mathsf{fill}\ x\ \mathsf{with}\ \bullet$) to remind us that the result of evaluating e should be memoized at x (Rules 49 and 50). If x is instead bound to junk, then we must be in the middle of evaluating another $\mathsf{force}(x)$, so we step to an Error state which halts the program (Rule 51).

Extending the type safety proof to handle memoized computations is straightforward. Continuation frame well-formedness is extended with Rule 52 for memoization frames. We must also update the definition of run-time contexts to include memoized location bindings, well-formed and terminal states to include Error, and store well-formedness to account for memoized locations:

Definition 7.4.1 (Run-Time Contexts)
A context Γ is *run-time* if the only bindings in Γ take the form $\mathcal{X}$, $x : \mathsf{box}_{\mathcal{T}}(\mathrm{C})$, $x : \mathsf{ref}(\mathrm{C})$, $x : \mathsf{cont}(\mathrm{C})$ or $x : \mathsf{comp}_{\mathcal{T}}(\mathrm{C})$.

Definition 7.4.2 (Machine State Well-formedness)
A machine state Ω is *well-formed* if either $\Omega = \mathsf{Error}$ or Ω is well-formed according to Definition 7.2.9.

Definition 7.4.3 (Terminal States)
A machine state Ω is *terminal* if either $\Omega = \mathsf{Error}$ or Ω is terminal according to Definition 7.2.13.

Definition 7.4.4 (Store Well-formedness)
A store ω is *well-formed*, denoted $\Gamma \vdash \omega\ [\mathcal{S}]$, if $\Gamma \vdash \omega\ [\mathcal{S}]$ according to Definition 7.3.2 and also:

- $\forall x : \mathsf{comp}_{\mathcal{T}}(\mathrm{C}) \in \Gamma.$ either $\omega(x) = \mathbf{?}$, or $\omega(x) = e$ and $\Gamma \vdash e : \mathrm{C}\ [\mathcal{S} \cup \mathcal{T}]$

7.5 Related Work

Safe Recursion Boudol [5] proposes a type system for safe recursion that, like mine, employs a backpatching semantics. Boudol's system tracks the *degrees* to which expressions depend on their free variables, where the degree to which e depends on x is 1 if x appears in a guarded position in e (*i.e.,* under an unapplied λ-abstraction), and 0 otherwise. What I call the "support" of an expression corresponds in Boudol's system to the set of variables on which the expression depends with degree 0. Thus, while there is no distinction between recursive and ordinary variables in Boudol's system, his equivalent of $\mathsf{saferec}(\mathcal{X} \triangleright x : \mathrm{C}.\, e)$ ensures that the evaluation of e will not dereference x by requiring that e depend on x with degree 1.

In my system an arrow type indicates the recursive variables that may be dereferenced when a function of that type is applied. An arrow type in Boudol's system indicates the degree to which the body of a function depends on its argument. Thus, $\mathrm{D} \xrightarrow{0} \mathrm{C}$ and $\mathrm{D} \xrightarrow{1} \mathrm{C}$ classify functions that are *strict* and *non-strict* in their arguments, respectively. As I discussed in Section 7.2.3, the ability to identify non-strict functions is especially important for purposes of separate compilation. For example, in order to typecheck the separate compilation scenario from Figure 5.11, it is necessary to know that the separately-compiled functors `F_Expr` and `F_Bind` are non-strict.

In contrast to my system, which requires the code from Figure 5.11 to be rewritten as shown in Figure 7.4, Boudol's system can typecheck the code in Figure 5.11 essentially as is. The reason is that function applications of the form $f(x)$ (where the argument is a variable) are treated as a special case in his semantics: while the expression "x" depends on the variable x with degree 0, the expression "$f(x)$" merely passes x to f without dereferencing it. This implies that ordinary λ-bound variables may be instantiated at run time with recursive variables. Thus, viewed in terms of my semantics, Boudol's system treats *all* variables as implicitly having box type.

The simplicity of Boudol's system is achieved at the expense of being rather conservative. In particular, a function application $f(e)$ is considered to depend on all the free variables of f with degree 0. Suppose that f is a curried function $\lambda y.\lambda z.e'$, where e' dereferences a recursive variable x. In Boudol's system, even a single application of f will be considered to depend on x with degree 0 and thus cannot appear unguarded in the recursive term defining x.

To address the limitations of Boudol's system, Hirschowitz and Leroy [35] propose a generalization of it, which they use as the target language for compiling a call-by-value mixin module calculus. Specifically, they extend Boudol's notion of degrees to be arbitrary integers: the degree to which e depends on x becomes, roughly, the number of unapplied λ-abstractions under which x appears in e. Thus, continuing the above example, the function $\lambda y.\lambda z.e'$ would depend on x with degree 2, so instantiating the first argument would only decrement that degree to 1, not 0.

Nevertheless, Hirschowitz and Leroy's system still suffers from a paucity of types. Consider the same curried function example, except where we let-bind $\lambda y.\lambda z.e'$ first instead of applying it directly: $\mathsf{let}\ f = \lambda y.\lambda z.e'\ \mathsf{in}\ f(e)$. The most precise degree-based type one can give to f when typing the body of the let is $\mathrm{C}_1 \xrightarrow{1} \mathrm{C}_2 \xrightarrow{0} \mathrm{C}_3$. This type tells us nothing about the degree to which f depends on the recursive variable x dereferenced by e'. Thus, Hirschowitz and Leroy's system must conservatively assume that $f(e)$ may dereference x. In contrast, my type system can assign f a more expressive type such as $\mathrm{C}_1 \xrightarrow{\emptyset} \mathrm{C}_2 \xrightarrow{\mathcal{X}} \mathrm{C}_3$, which would allow its first argument (but not its second) to be instantiated under the empty support.

I believe the let expression above is representative of code that one might want to write in the body of a recursive module, which suggests that my name-based approach is a more appropriate foundation for recursive modules. However, the weaknesses of the degree-based approaches are not necessarily problematic in the particular applications for which they were developed. For the purpose of compiling mixin modules, the primary feature required of Hirschowitz and Leroy's target language is the ability to link mutually recursive λ-abstractions that have been compiled separately. As I have illustrated in Section 7.2.3, my language supports this feature as well, via name abstractions.

Weak Polymorphism and Effect Systems There seems to be an analogy between the approaches discussed here for tracking safe recursion and the work on combining polymorphism and effects in the early days of ML. Boudol's 0-1 distinction is reminiscent of Tofte's distinction between imperative and applicative type variables [78]. Hirschowitz and Leroy's generalization of Boudol is similar to the idea of *weak polymorphism* [23] (implemented by MacQueen in earlier versions of the SML/NJ compiler), wherein a type variable α carries a numeric "strength" representing, roughly, the number of function applications required before a ref cell is created storing a value of type α. My system has ties to effect systems in the style of Talpin and Jouvelot [76], in which an arrow type indicates the set of effects that may occur when a function of that type is applied. For safe recursion, the effect in question is the dereferencing of an undefined recursive variable.

A common criticism leveled at both effect systems and weak polymorphism is that functional and imperative implementations of a polymorphic function have different types, and it is impossible to know which type to expect when designing a specification for a module separate from its implementation [82]. To a large extent, this criticism does not apply to my type system: names infect types *within* recursive modules, but the *external* interface of a module will be the same regardless of whether or not the module is implemented recursively. To ensure that certain recursive modules (like the one in Figure 7.5) are safe, however, one needs to observe that a general-purpose functor (like the `MakeSet` functor) is non-strict, and it is debatable whether the (non-)strictness of such a functor should be reflected in its specification. Choosing not to expose strictness information in the specification of a functor imposes fundamental limitations on how the functor can be used, not just in my system, but in any type system for safe recursion.

Strictness Analysis One can think of static detection of safe recursion as a kind of *non-strictness* analysis, in contrast to the well-known problem of *strictness* analysis [1]. Both problems are concerned with identifying whether an expression, such as the body of a function, will access the value of a particular variable when evaluated. Strictness analysis, however, is used as an optimization technique for lazy languages, in which any function may be conservatively classified as non-strict. In call-by-value languages, on the other hand, functions are strict by default—observing that a function is non-strict requires us to explicitly treat its argument as boxed and to show that applying the function will not unbox it. It is thus unclear how techniques from strictness analysis might be applied to the safe recursion problem.

Names The idea of using names in my type system is inspired by Nanevski's work on using a modal logic with names to model a "metaprogramming" language for symbolic computation [57]. (His use of names was in turn inspired by Pitts and Gabbay's FreshML [64].) Nanevski uses names to represent undefined symbols appearing inside expressions of a modal $\Box$ type. These expressions can be viewed as pieces of uncompiled syntax whose free names must be defined before they can be compiled.

My use of names is conceptually closer to Nanevski's more recent work on using names to model control effects for which there is a notion of handling [58]. One can think of the dereferencing of a recursive variable as an effect that is in some sense "handled" by the backpatching of the variable.

Formally, though, Nanevski's system is quite different, not least in that it does not employ any judgment of type equivalence modulo a support, which plays a critical role in my system.

Monadic Recursion There has been considerable work recently on adding effectful recursion to Haskell. Since effects in Haskell are isolated in monadic computations, adding a form of recursion over effectful expressions requires an understanding of how recursion interacts with monads. Erkök and Launchbury [17] propose a monadic fixed-point construct mfix for defining recursive computations in monads that satisfy a certain set of axioms. They later show how to use mfix to define a recursive form of Haskell's do construct [18]. Friedman and Sabry [21] argue that the backpatching semantics of recursion is fundamentally stateful, and thus defining a recursive computation in a given monad requires the monad to be combined with a state monad. This approach allows recursion in monads that do not obey the Erkök-Launchbury axioms, such as the continuation monad.

The primary goal of my type system is to statically ensure safe recursion in an impure call-by-value setting, and thus the work on recursive monadic computations for Haskell (which avoids any static analysis) is largely orthogonal to mine. Nevertheless, the dynamic semantics of my language borrows from the work of Moggi and Sabry [54], who give an operational semantics for the monadic metalanguage extended with the Friedman-Sabry mfix.

7.6 Directions for Future Work

In this chapter, I have studied the safe recursion problem at the level of the simply-typed λ-calculus. In order to incorporate the ideas from this calculus into a viable recursive module extension to ML, there are at least two significant issues that must be broached: (1) Do names and supports change the meaning of "principal" type schemes for ML terms? and (2) How should names be integrated into the ML module system? The first issue is clearly important, since type inference is one of ML's most notable features. The second issue is important as well, since the main motivation for safe recursion is to support more efficient recursive *modules*, not terms. I discuss these issues in Sections 7.6.1 and 7.6.2, respectively.

7.6.1 Names and Type Inference

The safe recursion language developed in this chapter requires explicit type and support annotations on λ-abstractions. In contrast, the ML language does not require the programmer to annotate function arguments (or any other variables) with their types; it infers the most general type scheme for a function argument by looking at how it is used in the function body. Is there a way to adapt Damas-Milner type inference [8] to infer principal type schemes in the presence of names?

Let us view inference as a procedure that takes an implicitly-typed external-language (EL) program and translates it into an explicitly-typed internal-language (IL) program. The main difficulty is that, if EL functions are not annotated with supports, there may be several incomparable ways to translate them. To take a simple example, say we are given a function `f` with input type C and output type D. If the body of `f` is well-typed under the empty support, then we can annotate `f` so that it has the type $\mathrm{C} \xrightarrow{\emptyset} \mathrm{D}$, but we can also annotate it to have the type $\mathrm{C} \xrightarrow{\mathcal{S}} \mathrm{D}$ for any well-formed support $\mathcal{S}$.

Name abstractions provide a potential solution to this problem. If the support of an arrow type is under-specified, a name abstraction can be used to generalize it. For example, if `f` has the form $\lambda x.e$, then we can generalize `f`'s type to $\forall \mathcal{X}.\mathrm{C} \xrightarrow{\mathcal{X}} \mathrm{D}$ by enclosing `f` in a name abstraction (*i.e.,* $\lambda\mathcal{X}.\lambda^{\mathcal{X}}x:\mathrm{C}.\,e$). This type scheme for `f` is *principal* in the sense that any valid type for `f` may be produced by instantiating $\mathcal{X}$ in the type scheme with a particular support. As with ML-style let-polymorphism, it is important that we only generalize the types of terms that are valuable (*i.e.,* pure and terminating)—otherwise, generalization could change the meaning of a term by suspending its effects.

While the generalized type for `f` is straightforward to understand, it is easy to construct terms for which the principal type scheme, or even the existence of one, is far from obvious. Consider a function `h`, defined as `fn (f,g) => fn x => f(fn y => x + g(x) + g(y))`. By simple inference reasoning, the use of addition implies that, modulo supports, `x` has type `int`, `g` has type `int` $\rightarrow$ `int`, and `f` has type (`int` $\rightarrow$ `int`) $\rightarrow$ `int`. One might therefore imagine inferring the following name-polymorphic type scheme for `h`:

$$\forall\mathcal{X}_1,\mathcal{X}_2,\mathcal{X}_3.(((\mathtt{int} \xrightarrow{\mathcal{X}_1} \mathtt{int}) \xrightarrow{\mathcal{X}_2} \mathtt{int}) \times (\mathtt{int} \xrightarrow{\mathcal{X}_1} \mathtt{int})) \xrightarrow{\mathcal{X}_3} (\mathtt{int} \xrightarrow{\mathcal{X}_2} \mathtt{int})$$

The support on `g`'s arrow type and the support on `f`'s argument's arrow type are matched up ($\mathcal{X}_1$); the support on the result arrow type equals the support on `f`'s arrow type ($\mathcal{X}_2$); and, since the body of `h` is a value, the support on `h`'s arrow type can be anything ($\mathcal{X}_3$).

Unfortunately, as complicated as it already is, this type scheme is not principal. In particular, assuming the existence of a name $\mathcal{X}$, the following is a valid type assignment for `h` which is not an instance of the above type scheme:

$$(((\mathtt{int} \xrightarrow{\emptyset} \mathtt{int}) \xrightarrow{\emptyset} \mathtt{int}) \times (\mathtt{int} \xrightarrow{\mathcal{X}} \mathtt{int})) \xrightarrow{\emptyset} (\mathtt{int} \xrightarrow{\mathcal{X}} \mathtt{int})$$

The point here is that it is fine for the support on `g`'s arrow to differ from the support on `f`'s argument's arrow, so long as the symmetric difference of the two supports is included in the result arrow. Correspondingly, here is a type scheme for `h` that I conjecture (but do not know) is principal:

$$\forall\mathcal{X}_1,\mathcal{X}_2,\mathcal{X}_3,\mathcal{X}_4,\mathcal{X}_5,\mathcal{X}_6.(((\mathtt{int} \xrightarrow{\{\mathcal{X}_1,\mathcal{X}_2\}} \mathtt{int}) \xrightarrow{\mathcal{X}_3} \mathtt{int}) \times (\mathtt{int} \xrightarrow{\{\mathcal{X}_1,\mathcal{X}_4\}} \mathtt{int})) \xrightarrow{\mathcal{X}_6} (\mathtt{int} \xrightarrow{\{\mathcal{X}_2,\mathcal{X}_3,\mathcal{X}_4,\mathcal{X}_5\}} \mathtt{int})$$

The supports on `g`'s arrow and `f`'s argument's arrow are permitted to differ ($\{\mathcal{X}_2,\mathcal{X}_4\}$), and the result arrow is allowed to contain arbitrary extra names ($\mathcal{X}_5$). The type assignment for `h` given above can be produced by instantiating the principal type scheme with $\mathcal{X}_4 = \mathcal{X}$ and $\mathcal{X}_i = \emptyset$ otherwise.

Determining whether principal type schemes exist in general in the presence of names is a key direction for future work. However, the possibly principal type scheme for `h` is a classic example of why people tend to be wary of effect inference. The inferred type is huge, hard to understand, and probably much more general than necessary. In fact, it is quite possible that the programmer will only use `h` at instances where all the $\mathcal{X}_i$ are set to $\emptyset$. I suspect therefore that, even if principal type schemes exist in the presence of names, it may nevertheless be a good idea to assume that λ-abstractions are annotated with the empty support unless specified otherwise, and to treat name abstraction and instantiation as explicit programmer-level operations.

7.6.2 Names and Modules

Another key direction for future work is to scale the type system presented in this chapter to the level of modules. In the process of preparing this thesis, I spent some time exploring this direction. I sketched the semantics and meta-theory of a safe recursive module construct, as an extension to the DCH language [12] (discussed in Section 2.3). The extension was rather complex and required global changes to the DCH type system. Some of this complexity appears to be unavoidable, but

I believe that some of it was also due to limitations of the DCH formalism. In this section, I will describe at a high level some of the issues that arise in building a safe recursive module extension to the type system of Chapters 3 and 4, and how we may deal with them. Working out the full details of such an extension remains important and non-trivial future work.

In order to scale the work of this chapter to the level of modules with type components, the first thing we must do is integrate names and supports into the core language of Chapter 3. This requires us to adapt the notion of type equivalence modulo a support to account for type constructors of higher kinds and singleton kinds. I believe this step is relatively straightforward, but it involves a global change to the language, which in turn necessitates a reproof of the entire Stone-Harper meta-theory!

The global change is essentially to add a support $[\mathcal{S}]$ to the end of every core-language judgment. The support of a judgment indicates the set of names that the derivation of the judgment can simply ignore. Thus, aside from the typing and type equivalence rules given in this chapter, all other rules will have the same support $\mathcal{S}$ in the premises as in the conclusion. Note that, because of singletons, supports must be added even to the *well-formedness* judgments for constructors and kinds. For instance, a constructor C has kind $\mathfrak{S}(\mathrm{D})$ if and only if C is equivalent to D at kind $\mathbf{T}$. If the two types are only equivalent modulo support $\mathcal{S}$, then C will only have kind $\mathfrak{S}(\mathrm{D})$ under support $\mathcal{S}$. Consequently, since the well-formedness of a singleton kind $\mathfrak{S}(\mathrm{C})$ depends on the well-formedness of C, $\mathfrak{S}(\mathrm{C})$ will only be well-formed under whatever support C requires to be well-formed.

The next step is to integrate names and supports into the module language. This may be done in an analogous way, adding supports to the end of every module and signature judgment form, with the interpretation that the names in the support may be ignored in the derivation of the judgment. On the surface, this seems straightforward, as the effect of "static purity" or "separability" tracked by the module system is completely orthogonal to the effect of dereferencing a recursive variable.

We must introduce a $\mathsf{saferec}(\mathcal{X} \triangleright \mathrm{X} : \mathbb{S}.\, \mathrm{M})$ module construct, as well as a $\mathsf{box}_{\mathcal{T}}(\mathrm{S})$ signature for classifying recursive module variables. Functors and functor signatures must be annotated with the support of their bodies. The rules for these constructs will combine their typing rules from Chapter 4 with the rules for their term-level counterparts given in this chapter. For example, a natural typing rule for total functors would be:

$$\frac{\Gamma, \mathrm{X}{:}\mathrm{S}' \vdash \mathrm{M} :_{\mathsf{P}} \mathrm{S}'' \ [\mathcal{S} \cup \mathcal{T}]}{\Gamma \vdash \lambda^{\mathcal{T}}_{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}'.\mathrm{M} :_{\mathsf{P}} \Pi^{\mathcal{T}}_{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}'.\mathrm{S}'' \ [\mathcal{S}]}$$

This is very similar to the typing rule for term-level λ-abstractions (Rule 5 of this chapter).

On closer inspection, however, an interesting problem arises with regard to the following question: what is the static part of a functor or functor signature bearing support $\mathcal{T}$, *i.e.,* how should we define $\mathsf{Fst}(\lambda^{\mathcal{T}}_{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}.\mathrm{M})$ and $\mathsf{Fst}(\Pi^{\mathcal{T}}_{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2)$? Since λ's at the constructor level do not have supports, the only obvious answer is to ignore the support $\mathcal{T}$, defining $\mathsf{Fst}(\lambda^{\mathcal{T}}_{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}.\mathrm{M})$ as $\lambda \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}).\mathsf{Fst}(\mathrm{M})$ and $\mathsf{Fst}(\Pi^{\mathcal{T}}_{\mathsf{tot}}\mathrm{X}{:}\mathrm{S}_1.\mathrm{S}_2)$ as $\Pi \mathrm{X}^{\mathsf{c}}{:}\mathsf{Fst}(\mathrm{S}_1).\mathsf{Fst}(\mathrm{S}_2)$.

To illustrate the problem with this answer, consider the functor $\mathrm{F} = \lambda^{\mathcal{X}}_{\mathsf{tot}}\_{:}1.[\mathtt{int} \xrightarrow{\mathcal{X}} \mathtt{int}]$, which takes unit argument and returns a module with a single type component, $\mathtt{int} \xrightarrow{\mathcal{X}} \mathtt{int}$. By the typing rule for functors given above, F can be assigned the signature $\mathrm{S} = \Pi^{\mathcal{X}}_{\mathsf{tot}}\_{:}1.[\![\mathfrak{S}(\mathtt{int} \xrightarrow{\emptyset} \mathtt{int})]\!]$ under the empty support. The point here is that the body is typechecked under the extended support $\{\mathcal{X}\}$, and under that support the type $\mathtt{int} \xrightarrow{\mathcal{X}} \mathtt{int}$ can be given the kind $\mathfrak{S}(\mathtt{int} \xrightarrow{\emptyset} \mathtt{int})$. Unfortunately, while F may have signature S under the empty support, $\mathsf{Fst}(\mathrm{F}) = \lambda\_{:}1.\mathtt{int} \xrightarrow{\mathcal{X}} \mathtt{int}$ does not have kind $\mathsf{Fst}(\mathrm{S}) = \Pi\_{:}1.\mathfrak{S}(\mathtt{int} \xrightarrow{\emptyset} \mathtt{int})$ under the empty support. This breaks an important hygienic property of the Fst function.

The problem is that the functor typing rule allows the well-formedness of the *static* part of F's body to depend on support $\mathcal{X}$, whereas the body of $\mathsf{Fst}(\mathrm{F})$ may not depend on that support. To address this problem, we must either redefine the Fst function or change the module typing judgment. It is not clear how to redefine Fst without first changing the core-language type system so that *constructor-level* functions may bear supports. While it is possible such an approach could work, it would require a more significant and less obvious change to the equational theory of types than what I described above.

An easier approach, I believe, is to rethink the module typing judgment $\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}\ [\mathcal{S}]$. Instead of ignoring the names in $\mathcal{S}$ in the whole typing derivation for M, suppose that we only ignore them when typechecking the *dynamic* part of M and that we typecheck the static part of M under empty support. For instance, the typing rules for atomic type and term modules might differ as follows:

$$\frac{\Gamma \vdash \mathrm{C} : \mathrm{K}\ [\emptyset]}{\Gamma \vdash [\mathrm{C}] :_{\mathrm{P}} [\![\mathrm{K}]\!]\ [\mathcal{S}]} \qquad \frac{\Gamma \vdash e : \mathrm{C}\ [\mathcal{S}]}{\Gamma \vdash [e] :_{\mathrm{P}} [\![\mathrm{C}]\!]\ [\mathcal{S}]}$$

The first rule would prevent the body of the functor F, $[\mathtt{int} \xrightarrow{\mathcal{X}} \mathtt{int}]$, from being assigned the signature $[\![\mathfrak{s}(\mathtt{int} \xrightarrow{\emptyset} \mathtt{int})]\!]$. More generally, the Fst function would regain the property that, if M has signature S under support $\mathcal{S}$, then $\mathsf{Fst}(\mathrm{M})$ has kind $\mathsf{Fst}(\mathrm{S})$ under the *empty* support.

To validate this re-interpretation of the module typing judgment, we also have to change the subsumption rule:

$$\frac{\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}'\ [\mathcal{S}] \quad \Gamma \vdash \mathrm{S}' \leq \mathrm{S}\ [\mathcal{S}]}{\Gamma \vdash \mathrm{M} :_\kappa \mathrm{S}\ [\mathcal{S}]}$$

This rule is problematic because, for example, it allows a module of signature $[\![\mathfrak{s}(\mathrm{C})]\!]$ to be coerced to the signature $[\![\mathfrak{s}(\mathrm{D})]\!]$, where C and D are only equivalent modulo support $\mathcal{S}$. An easy way to remedy this problem is to add to the subsumption rule an extra premise, $\Gamma \vdash \mathsf{Fst}(\mathrm{S}') \leq \mathsf{Fst}(\mathrm{S})\ [\emptyset]$, requiring that the static part of S′ match the static part of S under the empty support.[6]

Aside from this issue, I believe the integration of names and supports into my module type system should be fairly straightforward. It should be clear from this discussion, though, that supporting static detection of safe recursion involves a non-trivial, pervasive extension to the infrastructure and meta-theory of the language. It is not so clear whether the added benefit such an extension offers in terms of recursive module efficiency and reliability is worth the added complexity.

[6] In extending DCH, which did not have any notion of a Fst function, it was necessary to modify the signature subtyping judgment so that $\Gamma \vdash \mathrm{S}_1 \leq \mathrm{S}_2\ [\mathcal{S}]$ only ignored the names in $\mathcal{S}$ when matching the *dynamic* parts of S_1 against those of S_2.

Part III

Evolving the ML Module System

Chapter 8

Evolving the ML Internal Language

Parts I and II of this thesis have been devoted to achieving a clearer understanding of the ML module system and of the problem of extending ML with recursive modules. Based on this understanding, I described in Section 2.2 a proposal for unifying the existing variants of the ML module system, and in Section 5.4 a proposal for extending ML with recursive modules. It is time to make those proposals concrete.

In this and the next chapter, I will use the Harper-Stone interpretation of Standard ML [33] as the framework and starting point for defining a new, evolved dialect of ML. As described in Section 2.2.3, the Harper-Stone framework (hereafter, HS) defines SML by translating (or "elaborating") the programmer-level "external" language (EL) into an "internal" language (IL), which is defined by a type system. Following their approach, I will formalize my internal language in the present chapter and my external language in the next chapter.

My internal language is based very closely on the module type system I developed in Chapters 3 and 4 and extended in Chapter 6, which I will refer to as the "simplified IL." The differences between the simplified and actual IL's are mostly superficial. For instance, to facilitate the understanding of my new design by one who is already familiar with HS, I have chosen in most cases to use the HS conventions for naming metavariables rather than my own naming conventions from earlier in the thesis (*e.g.,* I use the metavariables *mod* and *sig* here instead of M and S to stand for modules and signatures).

There are a few non-trivial differences, however, which I discuss in Section 8.1. For those familiar with the details of Harper and Stone's formalism, I also discuss the ways in which my IL differs from theirs. Sections 8.2, 8.3 and 8.4 present the syntax, static semantics and dynamic semantics of my IL, respectively. As the IL is for the most part very similar to the simplified IL, I do not give an explicit typechecking algorithm or repeat the meta-theoretic development of Chapters 3, 4 and 6. Adapting them to the actual IL is completely straightforward.

8.1 Overview

8.1.1 Differences from the Simplified IL

Modulo naming conventions, the only substantive differences between the simplified and actual IL's are as follows.

First, instead of unlabeled pair kinds and pair signatures, whose components are indexed by position (π_1 or π_2), the actual IL provides n-ary *labeled* record kinds and record signatures, whose components are indexed by label. The generalization from pairs to n-ary records is a common, straightforward one. The generalization to labeled indexing is also straightforward.

It is important to understand that the labels on a record's components are not alpha-convertible. Each component of a record constructor or module has attached to it a label, by which it is referred to externally, and a variable, by which it is referred to internally, *i.e.,* by the other components defined after it in the record. Records are identified up to alpha-renaming of their bound *variables*, but not their labels.

As the subtyping relations on record kinds and signatures do not permit reordering or dropping of components, there is no significant semantic difference between these record kinds/signatures and the pair kinds/signatures of the simplified IL. The only differences are in the number of components in a record and how they are indexed. I use record kinds/signatures here instead of pairs primarily in order to model ML `structure`'s more faithfully and to hew closer to Harper-Stone.

Second, whereas in the simplified IL I modeled `type` and `val` bindings in structures as bindings of atomic modules [C] and [e], the actual IL allows one to bind types and terms directly as sub-components of record modules without encasing them in their own atomic modules. Again, this is not a matter of semantic importance, merely one of convenience.

Third, the actual IL provides a richer set of base types than the simplified IL. These include sum types, n-ary record types, mutable reference types, and an extensible sum type Tagged. The introduction and elimination forms for sums, records, and refs are standard. The type Tagged models the ML type of exceptions, `exn`. A value of type Tagged is a pair of a value *val* of type *con* and a "tag" of type *con* Tag. An ML exception binding like `exception E of int` results in the creation of a new tag of type $\mathsf{int}\,\mathsf{Tag}$ by invoking the primitive IL construct $\mathsf{new\_tag}[\mathsf{int}]$. Values *val* of type Tagged may be pattern-matched against tags of type *con* Tag by the primitive IL construct $\mathsf{iftagof}\ val\ \mathsf{is}\ val'\ \mathsf{then}\ val''\ \mathsf{else}\ exp$. This checks whether val' is the tag of *val*: if so, it passes the underlying value of *val* to the function val''; if not, it proceeds to evaluate the term exp'.

8.1.2 Differences from the Harper-Stone IL

The Harper-Stone IL is based on the Harper-Lillibridge (HL) type system [28] but only supports second-class modules. Thus, most of the differences between my IL and the Harper-Stone IL correspond to the differences between my system and HL's (see Section 2.2.1). For instance, HL model translucency through an explicit distinction between opaque and transparent type specifications in signatures, whereas I support translucency at the core-language level via singleton kinds. HL support only generative functors and impure sealing, whereas I support two forms of functors and two forms of sealing.

There are a variety of other small differences, including:

- The HS IL supports polymorphism only indirectly, *through* the module system, as a way of emphasizing the second-class character of polymorphism in ML. That is, to write a polymorphic function, one must write a functor that takes as input a module with one type component and returns as output a module with one monomorphic function component. My IL supports polymorphism directly in the core language in the style of System F_ω.

- The HS IL distinguishes between total and partial functions and functors, but what HS mean by "total" and "partial" is different from what those terms mean in my IL. A total function/functor in HS is one whose body is valuable, *i.e.,* effect-free and terminating. The bodies of total functors in my IL may have arbitrary effects as long as they are separable.

  HS track totality for two reasons. First, `datatype` constructors need to be given total arrow types so that one may observe that `datatype` constructor applications are valuable and thus eligible for polymorphic generalization. I address this issue in my language by giving `datatype` constructors *coercion* types, and by observing that coercion applications are always valuable. Second, HS model polymorphic functions as *total* functors in order to observe that the instantiation of a type abstraction is a valuable operation. In my language, I require the body of a type abstraction to be valuable, and therefore consider type instantiation to always be a valuable operation.

- HS characterize valuable expressions by means of a valuability judgment ($\Gamma \vdash exp \downarrow con$), whereas I simply characterize them as a syntactic subclass, which I write *vexp*. The only reason I can do this and HS cannot is that HS do not make a syntactic distinction between total and partial function application.

- The HS dynamic semantics employs a standard call-by-value left-to-right evaluation scheme. In contrast, my IL requires all sequencing of term-level operations to be made explicit through the use of a let expression. Thus, for instance, function application requires both the function and its argument to be values. As I illustrated in Sections 3.2 and 6.2, it is completely straightforward to use let expressions to define less restrictive versions of most constructs, wherein subterms are not required to be values and are evaluated left to right.

- For the purpose of defining its dynamic semantics, the HS IL introduces explicit syntactic classes of memory locations (*loc*) and extensible sum tags (*tag*), whereas I model these notions directly in terms of variables. On the other hand, HS join constructor variables, term variables and module variables under one syntactic class (*var*). For clarity, I make a syntactic distinction between *cvar*'s, *evar*'s and *mvar*'s, although I sometimes use *var* to signify any kind of variable.

- Lastly, my IL fixes a number of bugs and omissions in the HS IL.

8.2 IL Syntax

Figures 8.1–8.8 present the syntax of my internal language. I assume the following notational conventions:

- For any syntactic class (*e.g.,* labeled bindings, represented by the metavariable *lbnd*), I use the plural of the metavariable (*e.g., lbnds*) to denote a sequence of zero or more elements of the original class, separated by commas.

- Names for metavariables are complicated but logical. I use the letter p to mean *projectible*, the letter t to mean *transparent*, the letter c to mean *constructor*, the letter v to mean *valuable*, and the letter l to mean *labeled*. For instance, *tlcdec* denotes a transparent, labeled constructor declaration (see Figure 8.5). For those familiar with Harper-Stone, the metavariables *lbnds* and *ldecs* replace HS's *sbnds* and *sdecs*.

- Sequences of syntactic objects are assumed to be ordered, unless they appear inside curly braces, in which case they are identified up to reordering. The only instances of unordered sequences are in the syntax of record and sum types.

- I sometimes write $(phrase_i)_{i=1}^{n}$, where $phrase_i$ is some syntactic phrase that depends on i, to denote the sequence $phrase_1, \cdots, phrase_n$.

- I assume as a precondition for *syntactic* well-formedness that, in any sequence of bindings/declarations (*e.g.,* $lab_1 \triangleright var_1{:}sig_1, \cdots, lab_n \triangleright var_n{:}sig_n$), all the labels $lab_1, \cdots, lab_n$ are distinct, and all the variables $var_1, \cdots, var_n$ are distinct. Furthermore, I treat each var_i as being bound in the bindings/declarations that come after it. Sometimes I omit the variables and simply write $lab_1{:}sig_1, \cdots, lab_n{:}sig_n$, when I have no need to refer to the variable names.
- As in the simplified IL, I write $phrase'[phrase/var]$ to denote the capture-avoiding substitution of *phrase* for *var* in $phrase'$. When the *phrase* being substituted is a *pmod* (*i.e.,* a projectible module), $phrase'[pmod/mvar]$ is shorthand for $phrase'[\mathsf{Fst}(pmod)/mvar^{\mathsf{c}}]$, where $mvar^{\mathsf{c}}$ is the injection of *mvar* into the class of constructor variables. I will sometimes write $phrase'[phrase_i/var_i]_{i=1}^{n}$ as shorthand for $phrase'[phrase_1/var_1]\cdots[phrase_n/var_n]$.
- As in the simplified IL, I write $\rho(sig)$ as shorthand for $\rho(mvar).sig$ when $mvar^{\mathsf{c}} \notin \mathrm{FV}(sig)$.

$$
\begin{array}{rcll}
\mathit{knd} & ::= & \mathbf{T} & \text{kind of types} \\
 & | & \mathfrak{S}(\mathit{con}) & \text{singleton kind} \\
 & | & [\![\mathit{lcdecs}]\!] & \text{record kind} \\
 & | & \Pi(\mathit{cvar}{:}\mathit{knd}).\mathit{knd}' & \text{arrow kind} \\
 & & & \\
\mathit{bcon} & ::= & \mathsf{Int} \mid \mathsf{Float} \mid \cdots & \text{base types} \\
 & | & \{\mathit{rdecs}\} & \text{record type} \\
 & | & \mathit{con}\ \mathsf{Ref} & \text{reference type} \\
 & | & \mathit{con} \rightarrow \mathit{con}' & \text{function type} \\
 & | & \mathit{con} \leadsto \mathit{con}' & \text{coercion type} \\
 & | & \mathsf{Tagged} & \text{extensible sum type} \\
 & | & \mathit{con}\ \mathsf{Tag} & \text{exception-tag type} \\
 & | & \Sigma\{\mathit{rdecs}\} & \text{sum type} \\
 & | & \forall(\mathit{cvar}{:}\mathit{knd}).\mathit{con} & \text{universal type} \\
 & | & \exists(\mathit{cvar}{:}\mathit{knd}).\mathit{con} & \text{existential type} \\
 & | & \mathsf{maybe}(\mathit{con}) & \text{recursive variable type} \\
 & & & \\
\mathit{con} & ::= & \mathit{bcon} & \text{constructors of base kind } \mathbf{T} \\
 & | & \mathit{cvar} & \text{constructor variable} \\
 & | & \mu(\mathit{cvar}{:}\mathit{knd}).\mathit{con} & \text{iso-recursive constructor} \\
 & | & \lambda(\mathit{cvar}{:}\mathit{knd}).\mathit{con} & \text{constructor-level function} \\
 & | & \mathit{con}(\mathit{con}') & \text{constructor application} \\
 & | & [\mathit{lcbnds}] & \text{record of constructors} \\
 & | & \mathit{con}.\mathit{lab} & \text{record projection} \\
 & & & \\
\mathit{rdec} & ::= & \mathit{lab}{:}\mathit{con} & \text{record field type} \\
 & & & \\
\mathit{lcbnd} & ::= & \mathit{lab} \triangleright \mathit{cbnd} & \text{labeled constructor binding} \\
\mathit{cbnd} & ::= & \mathit{cvar}{=}\mathit{con} & \text{constructor binding} \\
 & & & \\
\mathit{lcdec} & ::= & \mathit{lab} \triangleright \mathit{cdec} & \text{labeled constructor declaration} \\
\mathit{cdec} & ::= & \mathit{cvar}{:}\mathit{knd} & \text{constructor declaration} \\
 & & & \\
\mathit{elim} & ::= & \bullet & \text{elimination head} \\
 & | & \mathit{elim}.\mathit{lab} & \text{projection} \\
 & | & \mathit{elim}(\mathit{con}) & \text{application} \\
\mathit{cpath} & ::= & \mathit{bcon} & \text{base type} \\
 & | & \mathit{elim}\{\mathit{cvar}\} & \text{path rooted at variable} \\
 & | & \mathit{recpath} & \text{recursive type path} \\
\mathit{recpath} & ::= & \mathit{elim}\{\mu(\mathit{cvar}{:}\mathit{knd}).\mathit{con}\} & \text{path rooted at } \mu\text{-constructor}
\end{array}
$$

Figure 8.1: IL Constructors and Kinds

val	$::=$	$scon$	constant
	$\mid$	$evar$	expression variable
	$\mid$	$\{rbnds\}$	record value
	$\mid$	$\mathsf{fix}_k\, fbnds\, \mathsf{end}$	recursive function
	$\mid$	$\mathsf{inj}_{lab}^{con}\ val$	sum injection
	$\mid$	$\mathsf{tag}(val, val')$	extensible type injection
	$\mid$	fold^{con}	fold coercion
	$\mid$	unfold^{con}	unfold coercion
	$\mid$	$\mathsf{fold}^{con}\langle\langle val\rangle\rangle$	coercion application
	$\mid$	$\Lambda(cvar{:}knd).vexp$	type abstraction
	$\mid$	$\mathsf{pack}\ [con, val]\ \mathsf{as}\ con'$	existential introduction
exp	$::=$	val	value
	$\mid$	$\pi_{lab}\ val$	record projection
	$\mid$	$val\ val'$	application
	$\mid$	$\mathsf{handle}\ exp\ \mathsf{with}\ val$	handle exception
	$\mid$	$\mathsf{raise}^{con}\ val$	raise exception
	$\mid$	$\mathsf{ref}\ val$	allocate new ref cell
	$\mid$	$\mathsf{get}\ val$	dereference a ref cell
	$\mid$	$\mathsf{set}\ (val, val')$	update a ref cell
	$\mid$	$val\langle\langle val'\rangle\rangle$	coercion application
	$\mid$	$val[con]$	type instantiation
	$\mid$	$\mathsf{case}\ val\ \mathsf{of}\ (lab_i{\mapsto}val_i)_{i=1}^{n}\ \mathsf{end}$	sum case analysis
	$\mid$	$\mathsf{new\_tag}[con]$	extend type Tagged
	$\mid$	$\mathsf{iftagof}\ val\ \mathsf{is}\ val'\ \mathsf{then}\ val''\ \mathsf{else}\ exp$	exception tag case analysis
	$\mid$	$\mathsf{let}\ [cvar, evar] = \mathsf{unpack}\ val\ \mathsf{in}\ (exp : con)$	existential elimination
	$\mid$	$\mathsf{let}\ cvar{=}con\ \mathsf{in}\ exp$	$cvar$-binding let expression
	$\mid$	$\mathsf{let}\ evar{=}exp'\ \mathsf{in}\ exp$	$evar$-binding let expression
	$\mid$	$\mathsf{let}\ mvar{=}mod\ \mathsf{in}\ (exp : con)$	$mvar$-binding let expression
	$\mid$	$\mathsf{rec}(evar{:}con.exp)$	recursive expression
	$\mid$	$\mathsf{fetch}(val)$	dereference a recursive variable
	$\mid$	$mod.lab$	module projection
	$\mid$	$\mathsf{pack}\ mod\ \mathsf{as}\ sig$	first-class module packing
$rbnd$	$::=$	$lab{=}val$	record field binding
$fbnd$	$::=$	$evar'(evar{:}con){:}con'{=}exp$	(recursive) function binding

Figure 8.2: IL Values and Expressions

mod	$::=$	$mvar$	module variable
	$\mid$	$[lbnds]$	structure
	$\mid$	$mod.lab$	projection from structure
	$\mid$	$\lambda^{\mathsf{tot}}(mvar{:}sig).mod$	total functor
	$\mid$	$\lambda^{\mathsf{par}}(mvar{:}sig){:}sig'.mod$	partial functor
	$\mid$	$mod^{\mathsf{tot}}(mod')$	total functor application
	$\mid$	$mod^{\mathsf{par}}(mod')$	partial functor application
	$\mid$	$mod :>_{\mathtt{P}} sig$	basic (weak) sealing
	$\mid$	$mod :>_{\mathtt{I}} sig$	impure (strong) sealing
	$\mid$	$\mathsf{unpack}\ exp\ \mathsf{as}\ sig$	first-class module unpacking
	$\mid$	$\mathsf{purify}(mod)$	purification of transparent module
	$\mid$	$\mathsf{let}\ mvar{=}mod\ \mathsf{in}\ (mod' : sig)$	let module
	$\mid$	$\mathsf{roll}(mod)$	rds introduction
	$\mid$	$\mathsf{unroll}(mod)$	rds elimination
	$\mid$	$\mathsf{fetch}(mod)$	dereference a recursive module variable
	$\mid$	$\mathsf{rec}(mvar{:}tsig.mod)$	recursive module
$lbnd$	$::=$	$lab \triangleright bnd$	labeled binding
bnd	$::=$	$cvar{=}con$	constructor binding
	$\mid$	$evar{=}exp$	expression binding
	$\mid$	$mvar{=}mod$	module binding
sig	$::=$	$[\![ldecs]\!]$	structure signature
	$\mid$	$\Pi^{\mathsf{tot}}(mvar{:}sig).sig'$	total functor signature
	$\mid$	$\Pi^{\mathsf{par}}(mvar{:}sig).sig'$	partial functor signature
	$\mid$	$\rho(mvar).sig$	recursively dependent signature (rds)
	$\mid$	$\mathsf{maybe}(sig)$	memoized computation signature
$ldec$	$::=$	$lab \triangleright dec$	labeled declaration
dec	$::=$	$cvar{:}knd$	constructor declaration
	$\mid$	$evar{:}con$	expression declaration
	$\mid$	$mvar{:}sig$	module declaration
var	$::=$	$cvar$	constructor variable
	$\mid$	$evar$	expression variable
	$\mid$	$mvar$	module variable

Figure 8.3: IL Modules and Signatures

$$
\begin{array}{rcl}
\mathit{vexp} & ::= & \mathit{val} \\
& | & \pi_{lab}\ \mathit{vexp} \\
& | & \mathit{vexp}\langle\!\langle \mathit{vexp}' \rangle\!\rangle \\
& | & \mathit{vexp}[\mathit{con}] \\
& | & \mathsf{inj}^{con}_{lab}\ \mathit{vexp} \\
& | & \mathsf{tag}(\mathit{vexp}, \mathit{vexp}') \\
& | & \mathit{vmod}.\mathit{lab}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{vmod} & ::= & \mathit{mvar} \\
& | & [\mathit{vlbnds}] \\
& | & \mathit{vmod}.\mathit{lab} \\
& | & \lambda^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{mod} \\
& | & \lambda^{\mathsf{par}}(\mathit{mvar}{:}\mathit{sig}){:}\mathit{sig}'.\mathit{mod} \\
& | & \mathsf{roll}(\mathit{vmod}) \\
& | & \mathsf{unroll}(\mathit{vmod}) \\
\\
\mathit{vlbnd} & ::= & \mathit{lab} \triangleright \mathit{vbnd} \\
\mathit{vbnd} & ::= & \mathit{cvar} {=} \mathit{vexp} \\
& | & \mathit{evar} {=} \mathit{con} \\
& | & \mathit{mvar} {=} \mathit{vmod}
\end{array}
$$

Figure 8.4: IL Valuable Expressions

$$
\begin{array}{rcl}
\mathit{pmod} & ::= & \mathit{mvar} \\
& | & [\mathit{plbnds}] \\
& | & \mathit{pmod}.\mathit{lab} \\
& | & \lambda^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{pmod} \\
& | & \lambda^{\mathsf{par}}(\mathit{mvar}{:}\mathit{sig}){:}\mathit{sig}'.\mathit{mod} \\
& | & \mathit{pmod}^{\mathsf{tot}}(\mathit{pmod}') \\
& | & \mathsf{roll}(\mathit{pmod}) \\
& | & \mathsf{unroll}(\mathit{pmod}) \\
& | & \mathsf{rec}(\mathit{mvar}{:}\mathit{tsig}.\mathit{mod}) \\
& | & \mathsf{fetch}(\mathit{pmod}) \\
\\
\mathit{plbnd} & ::= & \mathit{lab} \triangleright \mathit{pbnd} \\
\mathit{pbnd} & ::= & \mathit{cvar} {=} \mathit{con} \\
& | & \mathit{evar} {=} \mathit{exp} \\
& | & \mathit{mvar} {=} \mathit{pmod}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{tknd} & ::= & \mathfrak{S}(\mathit{con}) \\
& | & [\![\mathit{tlcdecs}]\!] \\
& | & \Pi(\mathit{cvar}{:}\mathit{knd}).\mathit{tknd}' \\
\\
\mathit{tlcdec} & ::= & \mathit{lab} \triangleright \mathit{tcdec} \\
\mathit{tcdec} & ::= & \mathit{cvar}{:}\mathit{tknd} \\
\\
\mathit{tsig} & ::= & [\![\mathit{tldecs}]\!] \\
& | & \Pi^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{tsig}' \\
& | & \Pi^{\mathsf{par}}(\mathit{mvar}{:}\mathit{sig}).\mathit{sig}' \\
& | & \rho(\mathit{mvar}).\mathit{tsig} \\
& | & \mathsf{maybe}(\mathit{tsig}) \\
\\
\mathit{tldec} & ::= & \mathit{lab} \triangleright \mathit{tdec} \\
\mathit{tdec} & ::= & \mathit{cvar}{:}\mathit{tknd} \\
& | & \mathit{evar}{:}\mathit{con} \\
& | & \mathit{mvar}{:}\mathit{tsig}
\end{array}
$$

Figure 8.5: IL Projectible Modules and Transparent Kinds/Signatures

$$
\begin{array}{rcl}
\mathit{sfknd} & ::= & \mathbf{T} \\
& | & [\![\mathit{lab}_1{:}\mathit{sfknd}_1, \cdots, \mathit{lab}_n{:}\mathit{sfknd}_n]\!] \\
& | & \mathit{sfknd} \rightarrow \mathit{sfknd}'
\end{array}
\qquad
\begin{array}{rcl}
\mathit{eknd} & ::= & \mathit{tknd} \\
& | & \mathbf{T} \\
& | & [\![\mathit{lab}_1{:}\mathit{eknd}_1, \cdots, \mathit{lab}_n{:}\mathit{eknd}_n]\!] \\
& | & \Pi(\mathit{cvar}{:}\mathit{sfknd}).\mathit{eknd}
\end{array}
$$

$$
\frac{}{\vdash \mu(\mathit{cvar}{:}\mathit{knd}).\mathit{con} \uparrow \mathit{knd}}
\qquad
\frac{\vdash \mathit{recpath} \uparrow [\![\cdots, \mathit{lab}{:}\mathit{knd}, \cdots]\!]}{\vdash \mathit{recpath}.\mathit{lab} \uparrow \mathit{knd}}
\qquad
\frac{\vdash \mathit{recpath} \uparrow \Pi(\mathit{cvar}{:}\mathit{knd}').\mathit{knd}}{\vdash \mathit{recpath}(\mathit{con}) \uparrow \mathit{knd}}
$$

$$
\frac{\vdash \mathit{recpath} \uparrow \mathbf{T} \qquad \mathit{recpath} = \mathit{elim}\{\mu(\mathit{cvar}{:}\mathit{eknd}).\mathit{con}\}}{\vdash \mathit{recpath}\ \textsf{expands}}
$$

$$
\textsf{expand}(\mathit{elim}\{\mu(\mathit{cvar}{:}\mathit{knd}).\mathit{con}\}) \stackrel{\text{def}}{=} \mathit{elim}\{\mathit{con}[\mu(\mathit{cvar}{:}\mathit{knd}).\mathit{con}/\mathit{cvar}]\}
$$

Figure 8.6: IL Expandable Kinds and Recursive Type Paths

$$
\frac{}{\vdash [\![\cdot]\!] \rightrightarrows [\![\mathit{cvar}{:}[\![\cdot]\!].\{\cdot\}]\!]}
$$

$$
\frac{\vdash [\![\mathit{ldecs}]\!] \rightrightarrows [\![\mathit{cvar}{:}[\![\mathit{lcdecs}]\!].\{\mathit{rdecs}\}]\!]}{\vdash [\![\mathit{lab}{\triangleright}\mathit{cvar}'{:}\mathit{knd}, \mathit{ldecs}]\!] \rightrightarrows [\![\mathit{cvar}{:}[\![\mathit{lab}{\triangleright}\mathit{cvar}'{:}\mathit{knd}, \mathit{lcdecs}]\!].\{\mathit{rdecs}\}[\mathit{cvar}.\mathit{lab}/\mathit{cvar}']]\!]}
$$

$$
\frac{\vdash [\![\mathit{ldecs}]\!] \rightrightarrows [\![\mathit{cvar}{:}[\![\mathit{lcdecs}]\!].\{\mathit{rdecs}\}]\!]}{\vdash [\![\mathit{lab}{\triangleright}\mathit{evar}{:}\mathit{con}, \mathit{ldecs}]\!] \rightrightarrows [\![\mathit{cvar}{:}[\![\mathit{lcdecs}]\!].\{\mathit{lab}{:}\mathit{con}, \mathit{rdecs}\}]\!]}
$$

$$
\frac{\vdash \mathit{sig} \rightrightarrows [\![\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}.\mathit{con}]\!] \qquad \vdash [\![\mathit{ldecs}]\!] \rightrightarrows [\![\mathit{cvar}{:}[\![\mathit{lcdecs}]\!].\{\mathit{rdecs}\}]\!]}{\vdash [\![\mathit{lab}{\triangleright}\mathit{mvar}{:}\mathit{sig}, \mathit{ldecs}]\!] \rightrightarrows [\![\mathit{cvar}{:}[\![\mathit{lab}{\triangleright}\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}, \mathit{lcdecs}]\!].\{\mathit{lab}{:}\mathit{con}, \mathit{rdecs}\}[\mathit{cvar}.\mathit{lab}/\mathit{mvar}^{\mathsf{c}}]]\!]}
$$

$$
\frac{\vdash \mathit{sig} \rightrightarrows [\![\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}.\mathit{con}]\!] \qquad \vdash \mathit{sig}' \rightrightarrows [\![\mathit{cvar}'{:}\mathit{knd}'.\mathit{con}']\!]}{\vdash \Pi^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{sig}' \rightrightarrows [\![\mathit{cvar}{:}\Pi(\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}).\mathit{knd}'.\forall(\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}).\mathit{con} \rightarrow \mathit{con}'[\mathit{cvar}(\mathit{mvar}^{\mathsf{c}})/\mathit{cvar}']]\!]}
$$

$$
\frac{\vdash \mathit{sig} \rightrightarrows [\![\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}.\mathit{con}]\!] \qquad \vdash \mathit{sig}' \rightrightarrows [\![\mathit{cvar}'{:}\mathit{knd}'.\mathit{con}']\!]}{\vdash \Pi^{\mathsf{par}}(\mathit{mvar}{:}\mathit{sig}).\mathit{sig}' \rightrightarrows [\![\mathit{cvar}{:}[\![\cdot]\!].\forall(\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}).\mathit{con} \rightarrow \exists(\mathit{cvar}'{:}\mathit{knd}').\mathit{con}']\!]}
$$

$$
\frac{\vdash \mathit{sig} \rightrightarrows [\![\mathit{cvar}{:}\mathit{knd}.\mathit{con}]\!]}{\vdash \rho(\mathit{mvar}).\mathit{sig} \rightrightarrows [\![\mathit{cvar}{:}\mathit{knd}.\mathit{con}[\mathit{cvar}/\mathit{mvar}^{\mathsf{c}}]]\!]}
$$

$$
\frac{\vdash \mathit{sig} \rightrightarrows [\![\mathit{cvar}{:}\mathit{knd}.\mathit{con}]\!]}{\vdash \textsf{maybe}(\mathit{sig}) \rightrightarrows [\![\mathit{cvar}{:}\mathit{knd}.\textsf{maybe}(\mathit{con})]\!]}
$$

$$
\langle\!|\mathit{sig}|\!\rangle \stackrel{\text{def}}{=} \exists(\mathit{cvar}{:}\mathit{knd}).\mathit{con}, \text{ where } \vdash \mathit{sig} \rightrightarrows [\![\mathit{cvar}{:}\mathit{knd}.\mathit{con}]\!]
$$

Figure 8.7: IL Definition of Package Type

$\mathsf{Can}(tknd)$	$\mathsf{Can}(\mathfrak{S}(con))$	$= con$
	$\mathsf{Can}(\llbracket tlcdecs \rrbracket)$	$= [\mathsf{Can}(tlcdecs)]$
	$\mathsf{Can}(\Pi(cvar{:}knd).tknd')$	$= \lambda(cvar{:}knd).\mathsf{Can}(tknd')$
$\mathsf{Can}(tlcdecs)$	$\mathsf{Can}(\cdot)$	$= \cdot$
	$\mathsf{Can}(lab \triangleright cvar{:}tknd, tlcdecs)$	$= lab \triangleright cvar{=}\mathsf{Can}(tknd), \mathsf{Can}(tlcdecs)$
$\mathsf{Fst}(sig)$	$\mathsf{Fst}(\llbracket ldecs \rrbracket)$	$= \llbracket \mathsf{Fst}(ldecs) \rrbracket$
	$\mathsf{Fst}(\Pi^{\mathsf{tot}}(mvar{:}sig).sig')$	$= \Pi(mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig)).\mathsf{Fst}(sig')$
	$\mathsf{Fst}(\Pi^{\mathsf{par}}(mvar{:}sig).sig')$	$= \llbracket \cdot \rrbracket$
	$\mathsf{Fst}(\rho(mvar).sig)$	$= \mathsf{Fst}(sig)$, where $mvar^{\mathsf{c}} \notin \mathrm{FV}(\mathsf{Fst}(sig))$
	$\mathsf{Fst}(\mathsf{maybe}(sig))$	$= \mathsf{Fst}(sig)$
$\mathsf{Fst}(ldecs)$	$\mathsf{Fst}(\cdot)$	$= \cdot$
	$\mathsf{Fst}(lab \triangleright cvar{:}knd, ldecs)$	$= lab \triangleright cvar{:}knd, \mathsf{Fst}(ldecs)$
	$\mathsf{Fst}(lab \triangleright evar{:}con, ldecs)$	$= \mathsf{Fst}(ldecs)$
	$\mathsf{Fst}(lab \triangleright mvar{:}sig, ldecs)$	$= lab \triangleright mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig), \mathsf{Fst}(ldecs)$
$\mathsf{Fst}(pmod)$	$\mathsf{Fst}(mvar)$	$= mvar^{\mathsf{c}}$
	$\mathsf{Fst}([plbnds])$	$= [\mathsf{Fst}(plbnds)]$
	$\mathsf{Fst}(pmod.lab)$	$= \mathsf{Fst}(pmod).lab$
	$\mathsf{Fst}(\lambda^{\mathsf{tot}}(mvar{:}sig).pmod)$	$= \lambda(mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig)).\mathsf{Fst}(pmod)$
	$\mathsf{Fst}(\lambda^{\mathsf{par}}(mvar{:}sig){:}sig'.mod)$	$= [\cdot]$
	$\mathsf{Fst}(pmod^{\mathsf{tot}}(pmod'))$	$= \mathsf{Fst}(pmod)(\mathsf{Fst}(pmod'))$
	$\mathsf{Fst}(\mathsf{roll}(pmod))$	$= \mathsf{Fst}(pmod)$
	$\mathsf{Fst}(\mathsf{unroll}(pmod))$	$= \mathsf{Fst}(pmod)$
	$\mathsf{Fst}(\mathsf{fetch}(pmod))$	$= \mathsf{Fst}(pmod)$
	$\mathsf{Fst}(\mathsf{rec}(mvar{:}tsig.mod))$	$= \mathsf{Can}(\mathsf{Fst}(tsig))$
$\mathsf{Fst}(plbnds)$	$\mathsf{Fst}(\cdot)$	$= \cdot$
	$\mathsf{Fst}(lab \triangleright cvar{=}con, plbnds)$	$= lab \triangleright cvar{=}con, \mathsf{Fst}(plbnds)$
	$\mathsf{Fst}(lab \triangleright mvar{=}pmod, plbnds)$	$= lab \triangleright mvar^{\mathsf{c}}{=}\mathsf{Fst}(pmod), \mathsf{Fst}(plbnds)$
	$\mathsf{Fst}(lab \triangleright evar{=}exp, plbnds)$	$= \mathsf{Fst}(plbnds)$
$\mathfrak{S}(con : knd)$	$\mathfrak{S}(con : \mathbf{T})$	$= \mathfrak{S}(con)$
	$\mathfrak{S}(con : \mathfrak{S}(con'))$	$= \mathfrak{S}(con)$
	$\mathfrak{S}(con : \llbracket lcdecs \rrbracket)$	$= \llbracket \mathfrak{S}(con : lcdecs) \rrbracket$
	$\mathfrak{S}(con : \Pi(cvar{:}knd).knd')$	$= \Pi(cvar{:}knd).\mathfrak{S}(con(cvar) : knd')$
$\mathfrak{S}(con : sig)$	$\mathfrak{S}(con : \llbracket ldecs \rrbracket)$	$= \llbracket \mathfrak{S}(con : ldecs) \rrbracket$
	$\mathfrak{S}(con : \Pi^{\mathsf{tot}}(mvar{:}sig).sig')$	$= \Pi^{\mathsf{tot}}(mvar{:}sig).\mathfrak{S}(con(mvar^{\mathsf{c}}) : sig')$
	$\mathfrak{S}(con : \Pi^{\mathsf{par}}(mvar{:}sig).sig')$	$= \Pi^{\mathsf{par}}(mvar{:}sig).sig'$
	$\mathfrak{S}(con : \rho(mvar).sig)$	$= \rho(\mathfrak{S}(con : sig[con/mvar^{\mathsf{c}}]))$
	$\mathfrak{S}(con : \mathsf{maybe}(sig))$	$= \mathsf{maybe}(\mathfrak{S}(con : sig))$
$\mathfrak{S}(con : ldecs)$	$\mathfrak{S}(con : \cdot)$	$= \cdot$
	$\mathfrak{S}(con :\ lab \triangleright cvar{:}knd, ldecs)$	$= lab{:}\mathfrak{S}(con.lab : knd),$ $\mathfrak{S}(con : ldecs[con.lab/cvar])$
	$\mathfrak{S}(con :\ lab \triangleright evar{:}con, ldecs)$	$= lab{:}con, \mathfrak{S}(con : ldecs)$
	$\mathfrak{S}(con :\ lab \triangleright mvar{:}sig, ldecs)$	$= lab{:}\mathfrak{S}(con.lab : sig),$ $\mathfrak{S}(con : ldecs[con.lab/mvar^{\mathsf{c}}])$

Figure 8.8: IL Meta-level Function Definitions

8.3 IL Static Semantics

As in the simplified IL, wherever "static" judgments $\mathcal{J}$ (*i.e.,* judgments about kinds, constructors or signatures, wherein the context is a sequence of *cdecs* instead of *decs*) are invoked with "dynamic" contexts *decs*, this is shorthand for the conjunction of $\vdash decs$ ok and $\mathsf{Fst}(decs) \vdash \mathcal{J}$.

Well-formed Declarations

$$\boxed{\vdash decs\ \mathbf{ok}}$$

$$\frac{}{\vdash \cdot\ \mathrm{ok}} \tag{8.1}$$

$$\frac{decs \vdash dec\ \mathrm{ok}}{\vdash decs, dec\ \mathrm{ok}} \tag{8.2}$$

$$\boxed{decs \vdash dec\ \mathbf{ok}}$$

$$\frac{decs \vdash knd : \mathsf{Kind} \qquad cvar \notin \mathrm{dom}(decs)}{decs \vdash cvar{:}knd\ \mathrm{ok}} \tag{8.3}$$

$$\frac{decs \vdash con : \mathbf{T} \qquad evar \notin \mathrm{dom}(decs)}{decs \vdash evar{:}con\ \mathrm{ok}} \tag{8.4}$$

$$\frac{decs \vdash sig : \mathsf{Sig} \qquad mvar \notin \mathrm{dom}(decs)}{decs \vdash mvar{:}sig\ \mathrm{ok}} \tag{8.5}$$

Well-formed Bindings

$$\boxed{decs \vdash bnd :_\kappa dec}$$

$$\frac{decs \vdash con : knd}{decs \vdash cvar{=}con :_{\mathrm{P}} cvar{:}knd} \tag{8.6}$$

$$\frac{decs \vdash exp : con}{decs \vdash evar{=}exp :_{\mathrm{P}} evar{:}con} \tag{8.7}$$

$$\frac{decs \vdash mod :_\kappa sig}{decs \vdash mvar{=}mod :_\kappa mvar{:}sig} \tag{8.8}$$

Well-formed Kinds

$$\boxed{cdecs \vdash knd : \mathsf{Kind}}$$

$$\frac{\vdash cdecs\ \mathrm{ok}}{cdecs \vdash \mathbf{T} : \mathsf{Kind}} \tag{8.9}$$

$$\frac{cdecs \vdash con : \mathbf{T}}{cdecs \vdash \mathfrak{S}(con) : \mathsf{Kind}} \tag{8.10}$$

$$\frac{cdecs \vdash lcdecs\ \mathrm{ok}}{cdecs \vdash [\![lcdecs]\!] : \mathsf{Kind}} \tag{8.11}$$

$$\frac{cdecs, cvar{:}knd \vdash knd' : \mathsf{Kind}}{cdecs \vdash \Pi(cvar{:}knd).knd' : \mathsf{Kind}} \tag{8.12}$$

Kind Equivalence

$$\boxed{\mathit{cdecs} \vdash \mathit{lcdecs} \equiv \mathit{lcdecs}'}$$

$$\frac{\vdash \mathit{cdecs}\ \mathrm{ok}}{\mathit{cdecs} \vdash \cdot\ \equiv\ \cdot} \tag{8.13}$$

$$\frac{\mathit{cdecs} \vdash \mathit{knd} \equiv \mathit{knd}' \qquad \mathit{cdecs}, \mathit{cvar}{:}\mathit{knd} \vdash \mathit{lcdecs}\ \equiv\ \mathit{lcdecs}'}{\mathit{cdecs} \vdash \mathit{lab}{\triangleright}\mathit{cvar}{:}\mathit{knd}, \mathit{lcdecs}\ \equiv\ \mathit{lab}{\triangleright}\mathit{cvar}{:}\mathit{knd}', \mathit{lcdecs}'} \tag{8.14}$$

$$\boxed{\mathit{cdecs} \vdash \mathit{knd} \equiv \mathit{knd}'}$$

$$\frac{\vdash \mathit{cdecs}\ \mathrm{ok}}{\mathit{cdecs} \vdash \mathbf{T} \equiv \mathbf{T}} \tag{8.15}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}' : \mathbf{T}}{\mathit{cdecs} \vdash \mathfrak{S}(\mathit{con}) \equiv \mathfrak{S}(\mathit{con}')} \tag{8.16}$$

$$\frac{\mathit{cdecs} \vdash \mathit{lcdecs}\ \equiv\ \mathit{lcdecs}'}{\mathit{cdecs} \vdash [\![\mathit{lcdecs}]\!] \equiv [\![\mathit{lcdecs}']\!]} \tag{8.17}$$

$$\frac{\mathit{cdecs} \vdash \mathit{knd}_1 \equiv \mathit{knd}_1' \qquad \mathit{cdecs}, \mathit{cvar}{:}\mathit{knd}_1 \vdash \mathit{knd}_2 \equiv \mathit{knd}_2'}{\mathit{cdecs} \vdash \Pi(\mathit{cvar}{:}\mathit{knd}_1).\mathit{knd}_2 \equiv \Pi(\mathit{cvar}{:}\mathit{knd}_1').\mathit{knd}_2'} \tag{8.18}$$

Kind Subtyping

$$\boxed{\mathit{cdecs} \vdash \mathit{lcdecs} \leq \mathit{lcdecs}'}$$

$$\frac{\vdash \mathit{cdecs}\ \mathrm{ok}}{\mathit{cdecs} \vdash \cdot\ \leq\ \cdot} \tag{8.19}$$

$$\frac{\mathit{cdecs} \vdash \mathit{knd} \leq \mathit{knd}' \qquad \mathit{cdecs}, \mathit{cvar}{:}\mathit{knd} \vdash \mathit{lcdecs}\ \leq\ \mathit{lcdecs}'}{\mathit{cdecs} \vdash \mathit{lab}{\triangleright}\mathit{cvar}{:}\mathit{knd}, \mathit{lcdecs}\ \leq\ \mathit{lab}{\triangleright}\mathit{cvar}{:}\mathit{knd}', \mathit{lcdecs}'} \tag{8.20}$$

$$\boxed{\mathit{cdecs} \vdash \mathit{knd} \leq \mathit{knd}'}$$

$$\frac{\vdash \mathit{cdecs}\ \mathrm{ok}}{\mathit{cdecs} \vdash \mathbf{T} \leq \mathbf{T}} \tag{8.21}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}' : \mathbf{T}}{\mathit{cdecs} \vdash \mathfrak{S}(\mathit{con}) \leq \mathfrak{S}(\mathit{con}')} \tag{8.22}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} : \mathbf{T}}{\mathit{cdecs} \vdash \mathfrak{S}(\mathit{con}) \leq \mathbf{T}} \tag{8.23}$$

$$\frac{\mathit{cdecs} \vdash \mathit{lcdecs}\ \leq\ \mathit{lcdecs}'}{\mathit{cdecs} \vdash [\![\mathit{lcdecs}]\!] \leq [\![\mathit{lcdecs}']\!]} \tag{8.24}$$

$$\frac{\begin{array}{c}\mathit{cdecs} \vdash \Pi(\mathit{cvar}{:}\mathit{knd}_1).\mathit{knd}_2 : \mathsf{Kind} \\ \mathit{cdecs} \vdash \mathit{knd}_1' \leq \mathit{knd}_1 \qquad \mathit{cdecs}, \mathit{cvar}{:}\mathit{knd}_1' \vdash \mathit{knd}_2 \leq \mathit{knd}_2'\end{array}}{\mathit{cdecs} \vdash \Pi(\mathit{cvar}{:}\mathit{knd}_1).\mathit{knd}_2 \leq \Pi(\mathit{cvar}{:}\mathit{knd}_1').\mathit{knd}_2'} \tag{8.25}$$

Well-formed Constructors

$$\boxed{cdecs \vdash con : knd}$$

$$\frac{\vdash cdecs \text{ ok} \qquad cvar{:}knd \in cdecs}{cdecs \vdash cvar : knd} \tag{8.26}$$

$$\frac{\vdash cdecs \text{ ok}}{cdecs \vdash \textsf{Tagged} : \mathbf{T}} \tag{8.27}$$

$$\frac{cdecs \vdash con : \mathbf{T}}{cdecs \vdash con\ \textsf{Ref} : \mathbf{T}} \tag{8.28}$$

$$\frac{cdecs \vdash con : \mathbf{T}}{cdecs \vdash con\ \textsf{Tag} : \mathbf{T}} \tag{8.29}$$

$$\frac{cdecs \vdash con : \mathbf{T} \qquad cdecs \vdash con' : \mathbf{T}}{cdecs \vdash con \rightarrow con' : \mathbf{T}} \tag{8.30}$$

$$\frac{cdecs \vdash con : \mathbf{T} \qquad cdecs \vdash con' : \mathbf{T}}{cdecs \vdash con \rightsquigarrow con' : \mathbf{T}} \tag{8.31}$$

$$\frac{cdecs \vdash con : \mathbf{T}}{cdecs \vdash \textsf{maybe}(con) : \mathbf{T}} \tag{8.32}$$

$$\frac{\vdash cdecs \text{ ok} \qquad \forall i \in 1..n: \ \ cdecs \vdash con_i : \mathbf{T}}{cdecs \vdash \{lab_1{:}con_1, \ldots, lab_n{:}con_n\} : \mathbf{T}} \tag{8.33}$$

$$\frac{\vdash cdecs \text{ ok} \qquad \forall i \in 1..n: \ \ cdecs \vdash con_i : \mathbf{T}}{cdecs \vdash \Sigma\{lab_1{:}con_1, \ldots, lab_n{:}con_n\} : \mathbf{T}} \tag{8.34}$$

$$\frac{cdecs, cvar{:}knd \vdash con : \mathbf{T}}{cdecs \vdash \forall(cvar{:}knd).con : \mathbf{T}} \tag{8.35}$$

$$\frac{cdecs, cvar{:}knd \vdash con : \mathbf{T}}{cdecs \vdash \exists(cvar{:}knd).con : \mathbf{T}} \tag{8.36}$$

$$\frac{cdecs, cvar{:}knd \vdash con : knd}{cdecs \vdash \mu(cvar{:}knd).con : knd} \tag{8.37}$$

$$\frac{cdecs \vdash lcbnds : lcdecs}{cdecs \vdash [lcbnds] : [\![lcdecs]\!]} \tag{8.38}$$

$$\frac{cdecs \vdash con : [\![\cdots, lab_i \triangleright cvar_i{:}knd_i, \cdots]\!]}{cdecs \vdash con.lab_i : knd_i[con.lab_j/cvar_j]_{j=1}^{i-1}} \tag{8.39}$$

$$\frac{cdecs, cvar{:}knd \vdash con : knd'}{cdecs \vdash \lambda(cvar{:}knd).con : \Pi(cvar{:}knd).knd'} \tag{8.40}$$

$$\frac{cdecs \vdash con : \Pi(cvar{:}knd').knd \qquad cdecs \vdash con' : knd'}{cdecs \vdash con(con') : knd[con'/cvar]} \tag{8.41}$$

$$\frac{cdecs \vdash con : \mathbf{T}}{cdecs \vdash con : \mathfrak{S}(con)} \tag{8.42}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} : [\![(\mathit{lab}_i \triangleright \mathit{cvar}_i{:}\mathit{knd}'_i)_{i=1}^n]\!] \qquad \forall i \in 1..n : \;\; \mathit{cdecs} \vdash \mathit{con}.\mathit{lab}_i : \mathit{knd}_i}{\mathit{cdecs} \vdash \mathit{con} : [\![(\mathit{lab}_i{:}\mathit{knd}_i)_{i=1}^n]\!]} \tag{8.43}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} : \Pi(\mathit{cvar}{:}\mathit{knd}).\mathit{knd}' \qquad \mathit{cdecs}, \mathit{cvar}{:}\mathit{knd} \vdash \mathit{con}(\mathit{cvar}) : \mathit{knd}''}{\mathit{cdecs} \vdash \mathit{con} : \Pi(\mathit{cvar}{:}\mathit{knd}).\mathit{knd}''} \tag{8.44}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} : \mathit{knd}' \qquad \mathit{cdecs} \vdash \mathit{knd}' \leq \mathit{knd}}{\mathit{cdecs} \vdash \mathit{con} : \mathit{knd}} \tag{8.45}$$

Constructor Equivalence

$$\boxed{\mathit{cdecs} \vdash \mathit{lcbnds} \equiv \mathit{lcbnds}' : \mathit{lcdecs}}$$

$$\frac{\vdash \mathit{cdecs}\ \mathrm{ok}}{\mathit{cdecs} \vdash \cdot \equiv \cdot : \cdot} \tag{8.46}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd} \qquad \mathit{cdecs}, \mathit{cvar}{:}\mathit{knd} \vdash \mathit{lcbnds} \equiv \mathit{lcbnds}' : \mathit{lcdecs}}{\mathit{cdecs} \vdash \mathit{lab} \triangleright \mathit{cvar}{=}\mathit{con}, \mathit{lcbnds} \equiv \mathit{lab} \triangleright \mathit{cvar}{=}\mathit{con}', \mathit{lcbnds}' : \mathit{lab} \triangleright \mathit{cvar}{:}\mathit{knd}, \mathit{lcdecs}} \tag{8.47}$$

$$\boxed{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} : \mathit{knd}}{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con} : \mathit{knd}} \tag{8.48}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con}' \equiv \mathit{con} : \mathit{knd}}{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd}} \tag{8.49}$$

$$\frac{\begin{array}{c}\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}' : \mathit{knd} \\ \mathit{cdecs} \vdash \mathit{con}' \equiv \mathit{con}'' : \mathit{knd}\end{array}}{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}'' : \mathit{knd}} \tag{8.50}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con}_1 : \mathfrak{S}(\mathit{con}) \qquad \mathit{cdecs} \vdash \mathit{con}_2 : \mathfrak{S}(\mathit{con})}{\mathit{cdecs} \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \mathfrak{S}(\mathit{con})} \tag{8.51}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \mathbf{T} \qquad \mathit{cdecs} \vdash \mathit{con}'_1 \equiv \mathit{con}'_2 : \mathbf{T}}{\mathit{cdecs} \vdash \mathit{con}_1 \rightarrow \mathit{con}'_1 \equiv \mathit{con}_2 \rightarrow \mathit{con}'_2 : \mathbf{T}} \tag{8.52}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con}_1 \equiv \mathit{con}_2 : \mathbf{T} \qquad \mathit{cdecs} \vdash \mathit{con}'_1 \equiv \mathit{con}'_2 : \mathbf{T}}{\mathit{cdecs} \vdash \mathit{con}_1 \rightsquigarrow \mathit{con}'_1 \equiv \mathit{con}_2 \rightsquigarrow \mathit{con}'_2 : \mathbf{T}} \tag{8.53}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}' : \mathbf{T}}{\mathit{cdecs} \vdash \mathit{con}\ \mathsf{Ref} \equiv \mathit{con}'\ \mathsf{Ref} : \mathbf{T}} \tag{8.54}$$

$$\frac{\mathit{cdecs} \vdash \mathit{con} \equiv \mathit{con}' : \mathbf{T}}{\mathit{cdecs} \vdash \mathit{con}\ \mathsf{Tag} \equiv \mathit{con}'\ \mathsf{Tag} : \mathbf{T}} \tag{8.55}$$

$$\frac{\vdash \mathit{cdecs}\ \mathrm{ok} \qquad \forall i \in 1..n : \;\; \mathit{cdecs} \vdash \mathit{con}_i \equiv \mathit{con}'_i : \mathbf{T}}{\mathit{cdecs} \vdash \{\mathit{lab}_1{:}\mathit{con}_1, \cdots, \mathit{lab}_n{:}\mathit{con}_n\} \equiv \{\mathit{lab}_1{:}\mathit{con}'_1, \cdots, \mathit{lab}_n{:}\mathit{con}'_n\} : \mathbf{T}} \tag{8.56}$$

$$\frac{\vdash \mathit{cdecs}\ \mathrm{ok} \qquad \forall i \in 1..n : \;\; \mathit{cdecs} \vdash \mathit{con}_i \equiv \mathit{con}'_i : \mathbf{T}}{\mathit{cdecs} \vdash \Sigma\{\mathit{lab}_1{:}\mathit{con}_1, \ldots, \mathit{lab}_n{:}\mathit{con}_n\} \equiv \Sigma\{\mathit{lab}_1{:}\mathit{con}'_1, \ldots, \mathit{lab}_n{:}\mathit{con}'_n\} : \mathbf{T}} \tag{8.57}$$

$$\frac{cdecs \vdash knd \equiv knd' \qquad cdecs, cvar{:}knd \vdash con \equiv con' : \mathbf{T}}{cdecs \vdash \forall(cvar{:}knd).con \equiv \forall(cvar{:}knd').con' : \mathbf{T}} \tag{8.58}$$

$$\frac{cdecs \vdash knd \equiv knd' \qquad cdecs, cvar{:}knd \vdash con \equiv con' : \mathbf{T}}{cdecs \vdash \exists(cvar{:}knd).con \equiv \exists(cvar{:}knd').con' : \mathbf{T}} \tag{8.59}$$

$$\frac{cdecs \vdash con \equiv con' : \mathbf{T}}{cdecs \vdash \mathsf{maybe}(con) \equiv \mathsf{maybe}(con') : \mathbf{T}} \tag{8.60}$$

$$\frac{cdecs \vdash knd \equiv knd' \qquad cdecs, cvar{:}knd \vdash con \equiv con' : knd}{cdecs \vdash \mu(cvar{:}knd).con \equiv \mu(cvar{:}knd').con' : knd} \tag{8.61}$$

$$\frac{cdecs \vdash lcbnds \equiv lcbnds' : lcdecs}{cdecs \vdash [lcbnds] \equiv [lcbnds'] : [\![lcdecs]\!]} \tag{8.62}$$

$$\frac{cdecs \vdash con \equiv con' : [\![\cdots, lab_i{\triangleright}cvar_i{:}knd_i, \cdots]\!]}{cdecs \vdash con.lab_i \equiv con'.lab_i : knd_i[con.lab_j/cvar_j]_{j=1}^{i-1}} \tag{8.63}$$

$$\frac{cdecs \vdash knd \equiv knd' \qquad cdecs, cvar{:}knd \vdash con \equiv con' : knd''}{cdecs \vdash \lambda(cvar{:}knd).con \equiv \lambda(cvar{:}knd').con' : \Pi(cvar{:}knd).knd''} \tag{8.64}$$

$$\frac{cdecs \vdash con_1 \equiv con_2 : \Pi(cvar{:}knd').knd \qquad cdecs \vdash con_1' \equiv con_2' : knd'}{cdecs \vdash con_1(con_1') \equiv con_2(con_2') : knd[con_1'/cvar]} \tag{8.65}$$

$$\frac{\begin{array}{c} cdecs \vdash con' : [\![(lab_i{\triangleright}cvar_i{:}knd_i')_{i=1}^{n}]\!] \qquad cdecs \vdash con'' : [\![(lab_i{\triangleright}cvar_i{:}knd_i'')_{i=1}^{n}]\!] \\ \forall i \in 1..n : \ \ cdecs \vdash con'.lab_i \equiv con''.lab_i : knd_i \end{array}}{cdecs \vdash con' \equiv con'' : [\![(lab_i{:}knd_i)_{i=1}^{n}]\!]} \tag{8.66}$$

$$\frac{\begin{array}{c} cdecs \vdash con_1 : \Pi(cvar{:}knd).knd_1 \qquad cdecs \vdash con_2 : \Pi(cvar{:}knd).knd_2 \\ cdecs, cvar{:}knd \vdash con_1(cvar) \equiv con_2(cvar) : knd' \end{array}}{cdecs \vdash con_1 \equiv con_2 : \Pi(cvar{:}knd).knd'} \tag{8.67}$$

$$\frac{cdecs \vdash con_1 \equiv con_2 : knd' \qquad cdecs \vdash knd' \leq knd}{cdecs \vdash con_1 \equiv con_2 : knd} \tag{8.68}$$

Well-formed Expressions

$\boxed{decs \vdash exp : con}$

$$\frac{\vdash decs \text{ ok}}{decs \vdash scon : \text{type}(scon)} \tag{8.69}$$

$$\frac{\vdash decs \text{ ok} \qquad evar{:}con \in decs}{decs \vdash evar : con} \tag{8.70}$$

$$\frac{decs \vdash val : con' \to con \qquad decs \vdash val' : con'}{decs \vdash val\ val' : con} \tag{8.71}$$

$$\frac{\forall i \in 1..n : \quad decs, (evar_j'{:}con_j \to con_j')_{j=1}^{n}, evar_i{:}con_i \vdash exp_i : con_i'}{decs \vdash \mathsf{fix}_k\ (evar_i'(evar_i{:}con_i){:}con_i'{=}exp_i)_{i=1}^{n}\ \mathsf{end} : con_k \to con_k'} \tag{8.72}$$

$$\frac{\vdash \mathit{decs}\ \mathrm{ok} \qquad \forall i \in 1..n:\ \ \mathit{decs} \vdash \mathit{val}_i : \mathit{con}_i}{\mathit{decs} \vdash \{\mathit{lab}_1{=}\mathit{val}_1, \cdots, \mathit{lab}_n{=}\mathit{val}_n\} : \{\mathit{lab}_1{:}\mathit{con}_1, \cdots, \mathit{lab}_n{:}\mathit{con}_n\}} \tag{8.73}$$

$$\frac{\mathit{decs} \vdash \mathit{val} : \{\mathit{rdecs}, \mathit{lab}{:}\mathit{con}, \mathit{rdecs}'\}}{\mathit{decs} \vdash \pi_{\mathit{lab}}\ \mathit{val} : \mathit{con}} \tag{8.74}$$

$$\frac{\mathit{decs} \vdash \mathit{exp} : \mathit{con} \qquad \mathit{decs} \vdash \mathit{val} : \mathsf{Tagged} \to \mathit{con}}{\mathit{decs} \vdash \mathsf{handle}\ \mathit{exp}\ \mathsf{with}\ \mathit{val} : \mathit{con}} \tag{8.75}$$

$$\frac{\mathit{decs} \vdash \mathit{val} : \mathsf{Tagged} \qquad \mathit{decs} \vdash \mathit{con} : \mathbf{T}}{\mathit{decs} \vdash \mathsf{raise}^{\mathit{con}}\ \mathit{val} : \mathit{con}} \tag{8.76}$$

$$\frac{\mathit{decs} \vdash \mathit{con} : \mathbf{T}}{\mathit{decs} \vdash \mathsf{new\_tag}[\mathit{con}] : \mathit{con}\ \mathsf{Tag}} \tag{8.77}$$

$$\frac{\mathit{decs} \vdash \mathit{val} : \mathit{con}}{\mathit{decs} \vdash \mathsf{ref}\ \mathit{val} : \mathit{con}\ \mathsf{Ref}} \tag{8.78}$$

$$\frac{\mathit{decs} \vdash \mathit{val} : \mathit{con}\ \mathsf{Ref}}{\mathit{decs} \vdash \mathsf{get}\ \mathit{val} : \mathit{con}} \tag{8.79}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{val} : \mathit{con}\ \mathsf{Ref} \\ \mathit{decs} \vdash \mathit{val}' : \mathit{con}\end{array}}{\mathit{decs} \vdash \mathsf{set}\,(\mathit{val}, \mathit{val}') : \{\}} \tag{8.80}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{recpath} : \mathbf{T} \qquad \vdash \mathit{recpath}\ \mathsf{expands}}{\mathit{decs} \vdash \mathsf{fold}^{\mathit{con}} : \mathsf{expand}(\mathit{recpath}) \leadsto \mathit{recpath}} \tag{8.81}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \mathit{recpath} : \mathbf{T} \qquad \vdash \mathit{recpath}\ \mathsf{expands}}{\mathit{decs} \vdash \mathsf{unfold}^{\mathit{con}} : \mathit{recpath} \leadsto \mathsf{expand}(\mathit{recpath})} \tag{8.82}$$

$$\frac{\mathit{decs} \vdash \mathit{val} : \mathit{con}' \leadsto \mathit{con} \qquad \mathit{decs} \vdash \mathit{val}' : \mathit{con}'}{\mathit{decs} \vdash \mathit{val}\langle\!\langle \mathit{val}' \rangle\!\rangle : \mathit{con}} \tag{8.83}$$

$$\frac{\mathit{decs}, \mathit{cvar}{:}\mathit{knd} \vdash \mathit{vexp} : \mathit{con}}{\mathit{decs} \vdash \Lambda(\mathit{cvar}{:}\mathit{knd}).\mathit{vexp} : \forall(\mathit{cvar}{:}\mathit{knd}).\mathit{con}} \tag{8.84}$$

$$\frac{\mathit{decs} \vdash \mathit{val} : \forall(\mathit{cvar}{:}\mathit{knd}).\mathit{con}' \qquad \mathit{decs} \vdash \mathit{con} : \mathit{knd}}{\mathit{decs} \vdash \mathit{val}[\mathit{con}] : \mathit{con}'[\mathit{con}/\mathit{cvar}]} \tag{8.85}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{con}' \equiv \exists(\mathit{cvar}{:}\mathit{knd}).\mathit{con}'' : \mathbf{T} \\ \mathit{decs} \vdash \mathit{con} : \mathit{knd} \qquad \mathit{decs} \vdash \mathit{val} : \mathit{con}''[\mathit{con}/\mathit{cvar}]\end{array}}{\mathit{decs} \vdash \mathsf{pack}\ [\mathit{con}, \mathit{val}]\ \mathsf{as}\ \mathit{con}' : \mathit{con}'} \tag{8.86}$$

$$\frac{\begin{array}{c}\mathit{decs} \vdash \mathit{val} : \exists(\mathit{cvar}{:}\mathit{knd}).\mathit{con}' \\ \mathit{decs}, \mathit{cvar}{:}\mathit{knd}, \mathit{evar}{:}\mathit{con}' \vdash \mathit{exp} : \mathit{con} \qquad \mathit{decs} \vdash \mathit{con} : \mathbf{T}\end{array}}{\mathit{decs} \vdash \mathsf{let}\,[\mathit{cvar}, \mathit{evar}] = \mathsf{unpack}\ \mathit{val}\,\mathsf{in}\,(\mathit{exp} : \mathit{con}) : \mathit{con}} \tag{8.87}$$

$$\frac{\mathit{decs} \vdash \mathit{con} \equiv \Sigma\{\mathit{lab}{:}\mathit{con}', \cdots\} : \mathbf{T} \qquad \mathit{decs} \vdash \mathit{val} : \mathit{con}'}{\mathit{decs} \vdash \mathsf{inj}_{\mathit{lab}}^{\mathit{con}}\ \mathit{val} : \mathit{con}} \tag{8.88}$$

$$\frac{decs \vdash val : \Sigma\{lab_1{:}con_1, \ldots, lab_n{:}con_n\} \qquad \forall i \in 1..n: \;\; decs \vdash val_i : con_i \to con}{decs \vdash \mathsf{case}\; val\; \mathsf{of}\; (lab_i{\mapsto}val_i)_{i=1}^{n}\; \mathsf{end} : con} \tag{8.89}$$

$$\frac{decs \vdash val : con\; \mathsf{Tag} \qquad decs \vdash val' : con}{decs \vdash \mathsf{tag}(val, val') : \mathsf{Tagged}} \tag{8.90}$$

$$\frac{\begin{array}{c} decs \vdash val : \mathsf{Tagged} \qquad decs \vdash val' : con\; \mathsf{Tag} \\ decs \vdash val'' : con \to con' \qquad decs \vdash exp : con' \end{array}}{decs \vdash \mathsf{iftagof}\; val\; \mathsf{is}\; val'\; \mathsf{then}\; val''\; \mathsf{else}\; exp : con'} \tag{8.91}$$

$$\frac{decs \vdash con : knd \qquad decs, cvar{:}knd \vdash exp : con'}{decs \vdash \mathsf{let}\; cvar{=}con\; \mathsf{in}\; exp : con'[con/cvar]} \tag{8.92}$$

$$\frac{decs \vdash exp : con \qquad decs, evar{:}con \vdash exp' : con'}{decs \vdash \mathsf{let}\; evar{=}exp\; \mathsf{in}\; exp' : con'} \tag{8.93}$$

$$\frac{decs \vdash mod :_{\kappa} sig \qquad decs, mvar{:}sig \vdash exp : con \qquad decs \vdash con : \mathbf{T}}{decs \vdash \mathsf{let}\; mvar{=}mod\; \mathsf{in}\; (exp : con) : con} \tag{8.94}$$

$$\frac{decs, evar{:}\mathsf{maybe}(con) \vdash exp : con}{decs \vdash \mathsf{rec}(evar{:}con.exp) : con} \tag{8.95}$$

$$\frac{decs \vdash val : \mathsf{maybe}(con)}{decs \vdash \mathsf{fetch}(val) : con} \tag{8.96}$$

$$\frac{decs \vdash pmod :_{\mathrm{P}} [\![\cdots, lab_i{\triangleright}var_i{:}con_i, \cdots]\!]}{decs \vdash pmod.lab_i : con_i[pmod.lab_j/var_j]_{j=1}^{i-1}} \tag{8.97}$$

$$\frac{decs \vdash mod :_{\kappa} sig}{decs \vdash \mathsf{pack}\; mod\; \mathsf{as}\; sig : \langle\!| sig |\!\rangle} \tag{8.98}$$

$$\frac{decs \vdash exp : con' \qquad decs \vdash con' \equiv con : \mathbf{T}}{decs \vdash exp : con} \tag{8.99}$$

Well-formed Signatures

$$\boxed{cdecs \vdash ldecs\; \mathbf{ok}}$$

$$\frac{\vdash cdecs\; \mathrm{ok}}{cdecs \vdash \cdot\; \mathrm{ok}} \tag{8.100}$$

$$\frac{cdecs, dec \vdash ldecs\; \mathrm{ok}}{cdecs \vdash lab{\triangleright}dec, ldecs\; \mathrm{ok}} \tag{8.101}$$

$$\boxed{cdecs \vdash sig : \mathsf{Sig}}$$

$$\frac{cdecs \vdash ldecs \text{ ok}}{cdecs \vdash [\![ldecs]\!] : \mathsf{Sig}} \tag{8.102}$$

$$\frac{cdecs \vdash sig : \mathsf{Sig} \qquad cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig) \vdash sig' : \mathsf{Sig}}{cdecs \vdash \Pi^{\mathsf{tot}}(mvar{:}sig).sig' : \mathsf{Sig}} \tag{8.103}$$

$$\frac{cdecs \vdash sig : \mathsf{Sig} \qquad cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig) \vdash sig' : \mathsf{Sig}}{cdecs \vdash \Pi^{\mathsf{par}}(mvar{:}sig).sig' : \mathsf{Sig}} \tag{8.104}$$

$$\frac{cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig) \vdash sig : \mathsf{Sig}}{cdecs \vdash \rho(mvar).sig : \mathsf{Sig}} \tag{8.105}$$

$$\frac{cdecs \vdash sig : \mathsf{Sig}}{cdecs \vdash \mathsf{maybe}(sig) : \mathsf{Sig}} \tag{8.106}$$

Signature Equivalence

$$\boxed{cdecs \vdash ldecs \ \equiv \ ldecs'}$$

$$\frac{\vdash cdecs \text{ ok}}{cdecs \vdash \cdot \ \equiv \ \cdot} \tag{8.107}$$

$$\frac{cdecs \vdash knd \equiv knd' \qquad cdecs, cvar{:}knd \vdash ldecs \ \equiv \ ldecs'}{cdecs \vdash lab{\triangleright}cvar{:}knd, ldecs \ \equiv \ lab{\triangleright}cvar{:}knd', ldecs'} \tag{8.108}$$

$$\frac{cdecs \vdash con \equiv con' : \mathbf{T} \quad cdecs \vdash ldecs \ \equiv \ ldecs'}{cdecs \vdash lab{\triangleright}evar{:}con, ldecs \ \equiv \ lab{\triangleright}evar{:}con', ldecs'} \tag{8.109}$$

$$\frac{cdecs \vdash sig \equiv sig' \quad cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig) \vdash ldecs \ \equiv \ ldecs'}{cdecs \vdash lab{\triangleright}mvar{:}sig, ldecs \ \equiv \ lab{\triangleright}mvar{:}sig', ldecs'} \tag{8.110}$$

$$\boxed{cdecs \vdash sig \equiv sig'}$$

$$\frac{cdecs \vdash ldecs \ \equiv \ ldecs'}{cdecs \vdash [\![ldecs]\!] \equiv [\![ldecs']\!]} \tag{8.111}$$

$$\frac{cdecs \vdash sig_1 \equiv sig_2 \quad cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig_1) \vdash sig'_1 \equiv sig'_2}{cdecs \vdash \Pi^{\mathsf{tot}}(mvar{:}sig_1).sig'_1 \equiv \Pi^{\mathsf{tot}}(mvar{:}sig_2).sig'_2} \tag{8.112}$$

$$\frac{cdecs \vdash sig_1 \equiv sig_2 \quad cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig_1) \vdash sig'_1 \equiv sig'_2}{cdecs \vdash \Pi^{\mathsf{par}}(mvar{:}sig_1).sig'_1 \equiv \Pi^{\mathsf{par}}(mvar{:}sig_2).sig'_2} \tag{8.113}$$

$$\frac{\begin{array}{c} cdecs \vdash \mathsf{Fst}(sig_1) \equiv \mathsf{Fst}(sig_2) \\ cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig_1) \vdash \mathfrak{S}(mvar^{\mathsf{c}} : sig_1) \equiv \mathfrak{S}(mvar^{\mathsf{c}} : sig_2) \\ cdecs \vdash \rho(mvar).sig_1 : \mathsf{Sig} \qquad cdecs \vdash \rho(mvar).sig_2 : \mathsf{Sig} \end{array}}{cdecs \vdash \rho(mvar).sig_1 \equiv \rho(mvar).sig_2} \tag{8.114}$$

$$\frac{cdecs \vdash sig_1 \equiv sig_2}{cdecs \vdash \mathsf{maybe}(sig_1) \equiv \mathsf{maybe}(sig_2)} \tag{8.115}$$

Signature Subtyping

$$\boxed{cdecs \vdash ldecs \;\leq\; ldecs'}$$

$$\frac{\vdash cdecs \text{ ok}}{cdecs \vdash \cdot \;\leq\; \cdot} \tag{8.116}$$

$$\frac{\begin{array}{c} cdecs \vdash knd \leq knd' \\ cdecs, cvar{:}knd \vdash ldecs \;\leq\; ldecs' \qquad cdecs, cvar{:}knd' \vdash ldecs' \text{ ok} \end{array}}{cdecs \vdash lab \triangleright cvar{:}knd, ldecs \;\leq\; lab \triangleright cvar{:}knd', ldecs'} \tag{8.117}$$

$$\frac{cdecs \vdash con \equiv con' : \mathbf{T} \qquad cdecs \vdash ldecs \leq ldecs'}{cdecs \vdash lab \triangleright evar{:}con, ldecs \;\leq\; lab \triangleright evar{:}con', ldecs'} \tag{8.118}$$

$$\frac{\begin{array}{c} cdecs \vdash sig \leq sig' \\ cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig) \vdash ldecs \;\leq\; ldecs' \qquad cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig') \vdash ldecs' \text{ ok} \end{array}}{cdecs \vdash lab \triangleright mvar{:}sig, ldecs \;\leq\; lab \triangleright mvar{:}sig', ldecs'} \tag{8.119}$$

$$\boxed{cdecs \vdash sig \leq sig'}$$

$$\frac{cdecs \vdash ldecs \;\leq\; ldecs'}{cdecs \vdash [\![ldecs]\!] \leq [\![ldecs']\!]} \tag{8.120}$$

$$\frac{cdecs \vdash sig_1 \equiv sig_2 \qquad cdecs, mvar^{\mathsf{c}}{:}sig_1 \vdash sig'_1 \leq sig'_2}{cdecs \vdash \Pi^{\mathsf{tot}}(mvar{:}sig_1).sig'_1 \leq \Pi^{\mathsf{tot}}(mvar{:}sig_2).sig'_2} \tag{8.121}$$

$$\frac{cdecs \vdash sig_1 \equiv sig_2 \qquad cdecs, mvar^{\mathsf{c}}{:}sig_1 \vdash sig'_1 \equiv sig'_2}{cdecs \vdash \Pi^{\mathsf{par}}(mvar{:}sig_1).sig'_1 \leq \Pi^{\mathsf{par}}(mvar{:}sig_2).sig'_2} \tag{8.122}$$

$$\frac{\begin{array}{c} cdecs \vdash \mathsf{Fst}(sig_1) \leq \mathsf{Fst}(sig_2) \\ cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig_1) \vdash \mathfrak{S}(mvar^{\mathsf{c}} : sig_1) \leq \mathfrak{S}(mvar^{\mathsf{c}} : sig_2) \\ cdecs \vdash \rho(mvar).sig_1 : \mathsf{Sig} \qquad cdecs \vdash \rho(mvar).sig_2 : \mathsf{Sig} \end{array}}{cdecs \vdash \rho(mvar).sig_1 \leq \rho(mvar).sig_2} \tag{8.123}$$

$$\frac{cdecs \vdash sig_1 \leq sig_2}{cdecs \vdash \mathsf{maybe}(sig_1) \leq \mathsf{maybe}(sig_2)} \tag{8.124}$$

Well-formed Modules

$$\boxed{decs \vdash lbnds :_{\kappa} ldecs}$$

$$\frac{\vdash decs \text{ ok}}{decs \vdash \cdot :_{\kappa} \cdot} \tag{8.125}$$

$$\frac{decs \vdash bnd :_{\kappa} dec \qquad decs, dec \vdash lbnds :_{\kappa} ldecs}{decs \vdash lab \triangleright bnd, lbnds :_{\kappa} lab \triangleright dec, ldecs} \tag{8.126}$$

$$\boxed{decs \vdash mod :_\kappa sig}$$

$$\frac{\vdash decs \text{ ok} \qquad mvar{:}sig \in decs}{decs \vdash mvar :_{\mathtt{P}} sig} \tag{8.127}$$

$$\frac{decs \vdash lbnds :_\kappa ldecs}{decs \vdash [lbnds] :_\kappa [\![ldecs]\!]} \tag{8.128}$$

$$\frac{decs \vdash pmod :_{\mathtt{P}} [\![\cdots, lab_i \triangleright var_i{:}sig_i, \cdots]\!]}{decs \vdash pmod.lab_i :_{\mathtt{P}} sig_i[pmod.lab_j/var_j]_{j=1}^{i-1}} \tag{8.129}$$

$$\frac{decs, mvar{:}sig \vdash mod :_{\mathtt{P}} sig'}{decs \vdash \lambda^{\mathsf{tot}}(mvar{:}sig).mod :_{\mathtt{P}} \Pi^{\mathsf{tot}}(mvar{:}sig).sig'} \tag{8.130}$$

$$\frac{decs, mvar{:}sig \vdash mod :_\kappa sig'}{decs \vdash \lambda^{\mathsf{par}}(mvar{:}sig){:}sig'.mod :_{\mathtt{P}} \Pi^{\mathsf{par}}(mvar{:}sig).sig'} \tag{8.131}$$

$$\frac{decs \vdash mod :_\kappa \Pi^{\mathsf{tot}}(mvar{:}sig').sig \qquad decs \vdash pmod' :_{\mathtt{P}} sig'}{decs \vdash mod^{\mathsf{tot}}(mod') :_\kappa sig[pmod'/mvar]} \tag{8.132}$$

$$\frac{decs \vdash mod :_\kappa \Pi^{\mathsf{par}}(mvar{:}sig').sig \qquad decs \vdash pmod' :_{\mathtt{P}} sig'}{decs \vdash mod^{\mathsf{par}}(mod') :_{\mathtt{I}} sig[pmod'/mvar]} \tag{8.133}$$

$$\frac{decs \vdash mod :_\kappa sig}{decs \vdash mod :>_{\mathtt{P}} sig :_\kappa sig} \tag{8.134}$$

$$\frac{decs \vdash mod :_\kappa sig}{decs \vdash mod :>_{\mathtt{I}} sig :_{\mathtt{I}} sig} \tag{8.135}$$

$$\frac{decs \vdash sig : \mathsf{Sig} \qquad decs \vdash exp : \langle\!|sig|\!\rangle}{decs \vdash \mathsf{unpack}\ exp\ \mathsf{as}\ sig :_{\mathtt{I}} sig} \tag{8.136}$$

$$\frac{decs \vdash mod :_\kappa tsig}{decs \vdash \mathsf{purify}(mod) :_{\mathtt{P}} tsig} \tag{8.137}$$

$$\frac{decs \vdash mod_1 :_\kappa sig_1 \qquad decs, mvar{:}sig_1 \vdash mod_2 :_\kappa sig \qquad decs \vdash sig : \mathsf{Sig}}{decs \vdash \mathsf{let}\ mvar{=}mod_1\ \mathsf{in}\ (mod_2 : sig) :_\kappa sig} \tag{8.138}$$

$$\frac{decs \vdash mod :_\kappa sig}{decs \vdash \mathsf{roll}(mod) :_\kappa \rho(sig)} \tag{8.139}$$

$$\frac{decs \vdash pmod :_{\mathtt{P}} \rho(mvar).sig}{decs \vdash \mathsf{unroll}(pmod) :_{\mathtt{P}} sig[pmod/mvar]} \tag{8.140}$$

$$\frac{decs \vdash mod :_\kappa \mathsf{maybe}(sig)}{decs \vdash \mathsf{fetch}(mod) :_\kappa sig} \tag{8.141}$$

$$\frac{decs, mvar{:}\mathsf{maybe}(tsig) \vdash mod :_{\mathtt{P}} tsig}{decs \vdash \mathsf{rec}(mvar{:}tsig.mod) :_{\mathtt{P}} tsig} \tag{8.142}$$

$$\frac{decs \vdash pmod :_{\mathtt{P}} sig}{decs \vdash pmod :_{\mathtt{P}} \mathfrak{S}(pmod : sig)} \tag{8.143}$$

$$\frac{decs \vdash mod :_{\mathtt{P}} sig}{decs \vdash mod :_{\mathtt{I}} sig} \tag{8.144}$$

$$\frac{decs \vdash mod :_{\kappa} sig' \qquad decs \vdash sig' \leq sig}{decs \vdash mod :_{\kappa} sig} \tag{8.145}$$

8.4 IL Dynamic Semantics

As in the simplified IL, the dynamic semantics is only defined for the core language. As the module language of the actual IL is almost identical to that of the simplified IL, adapting the phase-splitting translation of Sections 4.2.7 and 6.4 to the present setting is completely straightforward.

$$\begin{array}{rcll}
\Omega & ::= & (\omega, \mathcal{C}, exp) & \text{normal machine state} \\
& \mid & \mathsf{Error} & \text{error state} \\
\mathcal{C} & ::= & \bullet & \text{empty continuation stack} \\
& \mid & \mathcal{C} \circ \mathcal{F} & \text{non-empty continuation stack} \\
\mathcal{F} & ::= & \mathsf{let}\ evar{=}\bullet\ \mathsf{in}\ exp & \text{sequencing frame} \\
& \mid & \mathsf{rec}(evar{:}con.\bullet) & \text{recursive backpatching frame} \\
& \mid & \mathsf{handle}\bullet\mathsf{with}\ val & \text{exception handling frame}
\end{array}$$

Let $\mathcal{D}$ denote a continuation stack that does not contain any exception handling frames.

Small-Step Operational Semantics

$$\boxed{\Omega \mapsto \Omega'}$$

$$\frac{}{(\omega, \mathcal{C}, \pi_{lab}\ \{rbnds, lab{=}val, rbnds'\}) \mapsto (\omega, \mathcal{C}, val)} \tag{8.146}$$

$$\frac{fbnds = (evar'_i(evar_i{:}con_i){:}con'_i{=}exp_i)_{i=1}^{n}}{(\omega, \mathcal{C}, (\mathsf{fix}_k\ fbnds\ \mathsf{end})(val)) \mapsto (\omega, \mathcal{C}, exp_k[val/evar_k][\mathsf{fix}_i\ fbnds\ \mathsf{end}/evar'_i]_{i=1}^{n})} \tag{8.147}$$

$$\frac{}{(\omega, \mathcal{C}, \mathsf{handle}\ exp\ \mathsf{with}\ val) \mapsto (\omega, \mathcal{C} \circ \mathsf{handle}\bullet\mathsf{with}\ val, exp)} \tag{8.148}$$

$$\frac{}{(\omega, \mathcal{C} \circ (\mathsf{handle}\bullet\mathsf{with}\ val'), val) \mapsto (\omega, \mathcal{C}, val)} \tag{8.149}$$

$$\frac{}{(\omega, \mathcal{C} \circ (\mathsf{handle}\bullet\mathsf{with}\ val') \circ \mathcal{D}, \mathsf{raise}^{con}\ val) \mapsto (\omega, \mathcal{C}, val'\ val)} \tag{8.150}$$

$$\frac{}{(\omega, \mathcal{D}, \mathsf{raise}^{con}\ val) \mapsto \mathsf{Error}} \tag{8.151}$$

$$\frac{evar \notin \mathrm{dom}(\omega)}{(\omega, \mathcal{C}, \mathsf{ref}\ val) \mapsto (\omega[evar{\mapsto}val], \mathcal{C}, evar)} \tag{8.152}$$

$$\frac{evar \in \mathrm{dom}(\omega) \qquad \omega(evar) = val}{(\omega, \mathcal{C}, \mathsf{get}\ evar) \mapsto (\omega, \mathcal{C}, val)} \tag{8.153}$$

$$\frac{evar \in \mathrm{dom}(\omega)}{(\omega, \mathcal{C}, \mathsf{set}\ (evar, val)) \mapsto (\omega[evar{:=}val], \mathcal{C}, \{\})} \tag{8.154}$$

$$\frac{}{(\omega, \mathcal{C}, \mathsf{unfold}^{con} \langle\!\langle \mathsf{fold}^{con'} \langle\!\langle val \rangle\!\rangle \rangle\!\rangle) \mapsto (\omega, \mathcal{C}, val)} \tag{8.155}$$

$$\frac{}{(\omega, \mathcal{C}, (\Lambda(cvar{:}knd).vexp)[con]) \mapsto (\omega, \mathcal{C}, vexp[con/cvar])} \tag{8.156}$$

$$\frac{}{(\omega, \mathcal{C}, \mathsf{case}\ \mathsf{inj}^{con}_{lab_k}\ val\ \mathsf{of}\ (lab_i{\mapsto}val_i)^n_{i=1}\ \mathsf{end}) \mapsto (\omega, \mathcal{C}, val_k\ val)} \tag{8.157}$$

$$\frac{evar \notin \mathrm{dom}(\omega)}{(\omega, \mathcal{C}, \mathsf{new\_tag}[con]) \mapsto (\omega[evar{\mapsto}\mathbf{?}], \mathcal{C}, evar)} \tag{8.158}$$

$$\frac{}{(\omega, \mathcal{C}, \mathsf{iftagof}\ \mathsf{tag}(evar, val)\ \mathsf{is}\ evar\ \mathsf{then}\ val'\ \mathsf{else}\ exp) \mapsto (\omega, \mathcal{C}, val'\ val)} \tag{8.159}$$

$$\frac{evar \neq evar'}{(\omega, \mathcal{C}, \mathsf{iftagof}\ \mathsf{tag}(evar, val)\ \mathsf{is}\ evar'\ \mathsf{then}\ val'\ \mathsf{else}\ exp) \mapsto (\omega, \mathcal{C}, exp)} \tag{8.160}$$

$$\frac{}{\begin{array}{l}(\omega, \mathcal{C}, \mathsf{let}\ [cvar, evar] = \mathsf{unpack}\ (\mathsf{pack}\ [con, val]\ \mathsf{as}\ con')\ \mathsf{in}\ (exp : con'')) \mapsto \\ \quad (\omega, \mathcal{C}, exp[con/cvar][val/evar])\end{array}} \tag{8.161}$$

$$\frac{}{(\omega, \mathcal{C}, \mathsf{let}\ cvar{=}con\ \mathsf{in}\ exp) \mapsto (\omega, \mathcal{C}, exp[con/cvar])} \tag{8.162}$$

$$\frac{}{(\omega, \mathcal{C}, \mathsf{let}\ evar{=}exp'\ \mathsf{in}\ exp) \mapsto (\omega, \mathcal{C} \circ \mathsf{let}\ evar{=}\bullet\ \mathsf{in}\ exp, exp')} \tag{8.163}$$

$$\frac{}{(\omega, \mathcal{C} \circ \mathsf{let}\ evar{=}\bullet\ \mathsf{in}\ exp, val) \mapsto (\omega, \mathcal{C}, exp[val/evar])} \tag{8.164}$$

$$\frac{evar \notin \mathrm{dom}(\omega)}{(\omega, \mathcal{C}, \mathsf{rec}(evar{:}con.exp)) \mapsto (\omega[evar{\mapsto}\mathbf{?}], \mathcal{C} \circ \mathsf{rec}(evar{:}con.\bullet), exp)} \tag{8.165}$$

$$\frac{evar \in \mathrm{dom}(\omega)}{(\omega, \mathcal{C} \circ \mathsf{rec}(evar{:}con.\bullet), val) \mapsto (\omega[evar{:=}val], \mathcal{C}, val)} \tag{8.166}$$

$$\frac{evar \in \mathrm{dom}(\omega) \qquad \omega(evar) = val}{(\omega, \mathcal{C}, \mathsf{fetch}(evar)) \mapsto (\omega, \mathcal{C}, val)} \tag{8.167}$$

$$\frac{evar \in \mathrm{dom}(\omega) \qquad \omega(evar) = \mathbf{?}}{(\omega, \mathcal{C}, \mathsf{fetch}(evar)) \mapsto \mathsf{Error}} \tag{8.168}$$

Chapter 9

Evolving the ML External Language

With the internal language type system in hand, I can now define my external language. The main goal of the external language is to make life easier for the programmer. Thus, like Standard ML, which serves as the external language in Harper and Stone's framework (HS), my language supports type inference, pattern matching, `datatype` definitions, coercive signature matching, etc. These are features of convenience, not necessity. The main departure from this view of the external language is in the recursive module extension. As explained in Section 5.4, my elaboration translation for recursive modules supports data abstraction between recursive modules in a manner that is considerably more sophisticated than what is available in the internal language type system.

My external language is based closely on the syntax and semantics of Standard ML. It extends SML with support for higher-order modules, total and partial functors, basic and impure sealing, the packaging (and unpackaging) of modules as first-class values, recursive modules and recursively dependent signatures. However, it is not a conservative extension:

- There are a number of minor syntactic differences. For example, following O'Caml, I have chosen to replace `structure` and `functor` bindings with a single `module` binding form. In an actual implementation, it may be desirable to also support the SML syntax for purposes of backward compatibility, but I will ignore such concerns here.

- There are some features, such as type and structure sharing constraints on signatures, that I have chosen to interpret in a different manner than SML does, because I find the SML semantics to be problematic. I discuss the reasons for such instances of semantic divergence at the appropriate points in the chapter.

- There are some features of SML, such as `infix` declarations, that affect parsing but are not interesting from a semantic point of view. Following HS, my elaboration translation assumes that source-level programs have already been parsed into the abstract syntax of the external language, so I do not bother to account for such parsing-related features.

- There is one feature of Standard ML that I have chosen not to support at all in my language, namely the overloading of the equality operator. I assume that equality operators are defined for some fixed set of base types, and do not permit equality tests at other types. In my experience programming in SML and working on the TILT compiler, I have found overloaded equality to be of little use, yet it disproportionately complicates the semantics of the language. Incorporating a general form of overloading into ML (using *type classes*, for instance [25]) is a worthwhile endeavor, but it is beyond the scope of this thesis.

As in HS, the elaboration translation is defined by a set of judgments, whose inference rules are presented in declarative fashion but admit an algorithmic interpretation. The primary judgments have the form

$$\Gamma \vdash \textit{EL-phrase} \rightsquigarrow \textit{IL-phrase} : \textit{IL-classifier}$$

where the context Γ and *EL-phrase* may be viewed as inputs, and *IL-phrase* and *IL-classifier* as outputs. For some auxiliary judgments, the inputs and outputs are slightly less obvious. In Section 9.2, I describe how to interpret all the elaboration judgments, and I also stipulate several invariants that are assumed and maintained by the elaboration rules. Some of these invariants are described rather informally. For instance, if you see a module containing a field labeled "tag," then that module must be the output of elaborating an `exception` binding.

The most basic invariant, which I do state formally in Section 9.2, is soundness of the translation: if elaboration succeeds, it generates well-formed output (assuming in some cases certain preconditions on the input). Soundness is proved by straightforward induction on the elaboration rules. While I attest to having checked soundness for every rule, the sheer quantity and complexity of the elaboration rules prevents me from having total confidence that I have not overlooked some bug, serious or otherwise. In the process of preparing these rules, I uncovered many bugs in the original HS elaboration translation, and it is quite possible that my revisions and extensions have introduced new ones. Formalizing HS in the meta-logical framework Twelf [63], so that soundness may be checked mechanically, is a topic of current research at CMU.

Another desirable property is that elaboration be deterministic. Unfortunately, due to type inference this is not the case. For instance, an underconstrained EL phrase such as `fn x => x` may elaborate to an identity function at any type, $con \rightarrow con$—the generalization to the polymorphic type $\forall(cvar{:}\mathbf{T}).cvar \rightarrow cvar$ only occurs once the function is bound to a variable. HS argue informally that this element of nondeterminism is acceptable because valid outputs for the same input should only differ in ways that do not affect evaluation. As my extensions and revisions do not introduce any new sources of nondeterminism beyond what is already present in HS elaboration, I refer the reader to HS for further discussion of this point.

The chapter is structured as follows: In Section 9.1, I present the syntax of my external language. In Section 9.2, I give an overview of the judgments that comprise the elaboration translation, their interpretations and their soundness properties. In Section 9.3, I define the translation itself. Finally, in Section 9.4, I discuss the implementation of a variant of this external language that I undertook in the context of the TILT compiler.

9.1 EL Syntax

The syntax of the external language is shown in Figures 9.1 and 9.2. Optional elements are enclosed in angle brackets $\langle\cdots\rangle$. All optional choices are independent. Some points of note:

- I assume the existence of several (disjoint) base syntax classes, including: *base* (a fixed set of base types, including `int`, `float`, `unit`, etc.), *scon* (syntactic constants, same as in the IL), *tyvar* (type variables), *reclab* (record labels), *expid* (term identifiers), *conid* (type constructor identifiers), *modid* (module identifiers), and *sigid* (signature identifiers). For $id \in \{expid, conid, modid\}$, there is a corresponding "long" class *longid*, defined as

$$longid \;::=\; id \mid modid\,.\,longid$$

- Tuple types (of length n) are not supported explicitly, but are encodable as record types whose components are labeled 1 to n. (The class of *reclab*'s includes the natural numbers.)

$$
\begin{array}{rcl}
\mathit{ty} & ::= & \mathit{base} \\
& | & \mathit{tyvar} \\
& | & \texttt{\{}\mathit{reclab}_1 \texttt{ : } \mathit{ty}_1 \texttt{,} \cdots \texttt{,} \mathit{reclab}_n \texttt{ : } \mathit{ty}_n\texttt{\}} \\
& | & \mathit{ty} \texttt{ -> } \mathit{ty}' \\
& | & \texttt{(}\mathit{ty}_1 \texttt{,} \cdots \texttt{,} \mathit{ty}_n\texttt{)}\ \langle \mathit{modexp}\,\texttt{.}\rangle \mathit{conid} \\
\\
\mathit{expr} & ::= & \mathit{scon} \\
& | & \mathit{expid} \\
& | & \mathit{modexp}\,\texttt{.}\,\mathit{expid} \\
& | & \texttt{\{}\mathit{reclab}_1 \texttt{ = } \mathit{expr}_1 \texttt{,} \cdots \texttt{,} \mathit{reclab}_n \texttt{ = } \mathit{expr}_n\texttt{\}} \\
& | & \texttt{let } \mathit{bindings} \texttt{ in } \mathit{expr} \texttt{ end} \\
& | & \mathit{expr}\ \mathit{expr}' \\
& | & \mathit{expr} \texttt{ : } \mathit{ty} \\
& | & \mathit{expr} \texttt{ handle } \mathit{match} \\
& | & \texttt{raise } \mathit{expr} \\
& | & \texttt{fn } \mathit{match} \\
& | & \texttt{pack } \mathit{modexp} \texttt{ as } \mathit{longconid} \\
\\
\mathit{bindings} & ::= & \cdot \\
& | & \mathit{binding}\ \langle\texttt{;}\rangle\ \mathit{bindings} \\
\mathit{binding} & ::= & \texttt{val (}\mathit{tyvar}_1 \texttt{,} \cdots \texttt{,} \mathit{tyvar}_n\texttt{)}\ \mathit{pat} \texttt{ = } \mathit{expr} \\
& | & \texttt{fun (}\mathit{tyvar}_1 \texttt{,} \cdots \texttt{,} \mathit{tyvar}_n\texttt{)}\ \mathit{funbinds} \\
& | & \texttt{open } \mathit{longmodid}_1\ \cdots\ \mathit{longmodid}_n \\
& | & \texttt{exception } \mathit{expid} \\
& | & \texttt{exception } \mathit{expid} \texttt{ of } \mathit{ty} \\
& | & \texttt{exception } \mathit{expid} \texttt{ = } \mathit{longexpid} \\
& | & \texttt{local } \mathit{bindings}_1 \texttt{ in } \mathit{bindings}_2 \texttt{ end} \\
& | & \texttt{type } \mathit{tybind} \\
& | & \texttt{datatype } \mathit{datbinds} \\
& | & \texttt{datatype } \mathit{conid} \texttt{ = datatype } \mathit{longconid} \\
& | & \texttt{packtype } \mathit{conid} \texttt{ = } \mathit{sigexp} \\
& | & \texttt{signature } \mathit{sigid} \texttt{ = } \mathit{sigexp} \\
& | & \texttt{module } \mathit{modid} \texttt{ = } \mathit{modexp} \\
\\
\mathit{funbinds} & ::= & \mathit{expid} \texttt{ = fn } \mathit{match}\ \langle\texttt{and } \mathit{funbinds}\rangle \\
\mathit{tybind} & ::= & \texttt{(}\mathit{tyvar}_1 \texttt{,} \cdots \texttt{,} \mathit{tyvar}_n\texttt{)}\ \mathit{conid} \texttt{ = } \mathit{ty} \\
\mathit{branches} & ::= & \mathit{expid}\ \langle\texttt{of } \mathit{ty}\rangle\ \langle\texttt{| } \mathit{branches}\rangle \\
\mathit{datbind} & ::= & \texttt{(}\mathit{tyvar}_1 \texttt{,} \cdots \texttt{,} \mathit{tyvar}_n\texttt{)}\ \mathit{conid} \texttt{ = } \mathit{branches} \\
\mathit{datbinds} & ::= & \mathit{datbind}_1\ \langle\texttt{and }\cdots\texttt{ and } \mathit{datbind}_n\rangle \\
& & \langle\texttt{withtype } \mathit{tybind}_1 \texttt{ and } \cdots \texttt{ and } \mathit{tybind}_m\rangle
\end{array}
$$

Figure 9.1: Syntax of the External Language

$$
\begin{array}{rcl}
\mathit{match} & ::= & \mathit{mrule} \\
& | & \mathit{mrule}\ \texttt{|}\ \mathit{match} \\
\mathit{mrule} & ::= & \mathit{pat}\ \texttt{=>}\ \mathit{expr} \\
\mathit{pat} & ::= & \mathit{scon} \\
& | & \mathit{longexpid} \\
& | & \texttt{\_} \\
& | & \mathit{pat}\ \texttt{:}\ \mathit{ty} \\
& | & \mathit{longexpid}\ \mathit{pat} \\
& | & \texttt{\{}\mathit{reclab}_1\ \texttt{=}\ \mathit{pat}_1\texttt{,}\cdots\texttt{,}\mathit{reclab}_n\ \texttt{=}\ \mathit{pat}_n\langle\texttt{,}\ \texttt{...}\rangle\texttt{\}} \\
& | & \mathit{pat}_1\ \texttt{as}\ \mathit{pat}_2 \\
& | & \texttt{ref}\ \mathit{pat} \\
\\
\mathit{sigexp} & ::= & \mathit{sigid} \\
& | & \texttt{sig}\ \mathit{specs}\ \texttt{end} \\
& | & \texttt{functor}\ \texttt{(}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{->}\ \mathit{sigexp}' \\
& | & \texttt{functor}\ \texttt{(}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{->>}\ \mathit{sigexp}' \\
& | & \texttt{rec}\ \texttt{(}\mathit{modid}\texttt{)}\ \mathit{sigexp} \\
& | & \mathit{sigexp}\ \texttt{where}\ \texttt{type}\ \texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{longconid}\ \texttt{=}\ \mathit{ty} \\
\mathit{specs} & ::= & \cdot \\
& | & \mathit{spec}\ \mathit{specs} \\
\mathit{spec} & ::= & \texttt{val}\ \texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{expid}\ \texttt{:}\ \mathit{ty} \\
& | & \texttt{type}\ \texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{conid} \\
& | & \texttt{type}\ \texttt{(}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{conid}\ \texttt{=}\ \mathit{ty} \\
& | & \texttt{datatype}\ \mathit{datbinds} \\
& | & \texttt{datatype}\ \mathit{conid}\ \texttt{=}\ \texttt{datatype}\ \mathit{longconid} \\
& | & \texttt{packtype}\ \mathit{conid}\ \texttt{=}\ \mathit{sigexp} \\
& | & \texttt{exception}\ \mathit{expid} \\
& | & \texttt{exception}\ \mathit{expid}\ \texttt{of}\ \mathit{ty} \\
& | & \texttt{module}\ \mathit{modid}\ \texttt{:}\ \mathit{sigexp} \\
& | & \texttt{include}\ \mathit{sigexp} \\
& | & \mathit{specs}\ \texttt{sharing}\ \texttt{type}\ \mathit{longconid}_1\ \texttt{=}\ \mathit{longconid}_2 \\
& | & \mathit{specs}\ \texttt{sharing}\ \mathit{longmodid}_1\ \texttt{=}\ \mathit{longmodid}_2 \\
\\
\mathit{seal} & ::= & \texttt{:} \\
& | & \texttt{:>} \\
& | & \texttt{:>>} \\
\mathit{modexp} & ::= & \mathit{modid} \\
& | & \texttt{struct}\ \mathit{bindings}\ \texttt{end} \\
& | & \mathit{modexp}\texttt{.}\mathit{modid} \\
& | & \texttt{functor}\ \texttt{(}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{->}\ \mathit{modexp} \\
& | & \texttt{functor}\ \texttt{(}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \texttt{->>}\ \mathit{modexp} \\
& | & \mathit{modexp}_1\ \texttt{(}\mathit{modexp}_2\texttt{)} \\
& | & \texttt{let}\ \mathit{bindings}\ \texttt{in}\ \mathit{modexp}\ \texttt{end} \\
& | & \mathit{modexp}\ \mathit{seal}\ \mathit{sigexp} \\
& | & \texttt{unpack}\ \mathit{expr}\ \texttt{as}\ \mathit{longconid} \\
& | & \texttt{rec}\ \texttt{(}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \mathit{modexp}
\end{array}
$$

Figure 9.2: Syntax of the External Language (continued)

- Unlike in SML, types, terms and modules may be projected from arbitrary module expressions *modexp*, not just from *longmodid*'s. While the elaboration translation places restrictions on which modules may be projected from, they are not simple syntactic restrictions. Nevertheless, there are several instances in the syntax where a type is required to have the form *longconid* or a module is required to have the form *longmodid* (*e.g.,* in `sharing` constraints on signatures). The typical reason for making such a restriction is to ensure that the translation of the type or module has a certain form (*e.g.,* see Rule 9.25).

- In much the same way that recursive and sum types are accessible to the SML programmer only through the `datatype` mechanism, existential types are accessible to the programmer of my language only through the `packtype` mechanism. The binding `packtype` *conid* `=` *sigexp* elaborates to a module containing three components: (1) an abstract type *conid*, implemented internally as $\langle\!|sig|\!\rangle$, where *sig* is the IL translation of *sigexp*, (2) a functor named *conid*_pack, which takes modules of signature *sig* and packages them as values of type *conid*, and (3) a functor named *conid*_ unpack, which unpackages values of type *conid* into modules of signature *sig*. Correspondingly, the EL term `pack` *modexp* `as` *longconid* and the EL module `unpack` *expr* `as` *longconid* package *modexp* and unpackage *expr*, respectively, by applying to them the "pack" and "unpack" functors located in the module defining *longconid*. The purpose of hiding the implementation of *conid* behind an abstract interface is to permit the introduction of package types into the language without affecting the type inference/unification algorithm.

- Like HS and unlike the Definition of SML, I do not distinguish between top-level bindings and module bindings. There is only one kind of *binding* (HS use the term *strdec* instead). Following O'Caml, I replace `structure` and `functor` bindings with a single `module` binding form. I also introduce anonymous functor expressions `functor (`*modid* `:` *sigexp*`) ->` *modexp* and `functor (`*modid* `:` *sigexp*`) ->>` *modexp*, where the former denotes a total functor and the latter a partial functor. I similarly distinguish between total and partial functor *signatures* by writing the former with a "`->`" arrow and the latter with with a "`->>`" arrow. SML's `functor` binding may be encoded as a `module` binding of a partial `functor` expression.

- The language supports three forms of sealing: transparent (`:`), basic (`:>`) and impure (`:>>`). Transparent sealing has precisely the same semantics as in SML (see the note on Rule 9.52 for details). The latter two correspond directly to $:>_{\mathtt{P}}$ and $:>_{\mathtt{I}}$ in the IL, respectively. The notation is perhaps slightly unfortunate here. As explained in Section 2.2.1, although basic sealing is denoted here in the same way that opaque sealing is denoted in SML, SML's opaque sealing is in fact closer semantically to impure sealing (`:>>`). Nonetheless, for uniformity, I have chosen to mark the distinction between basic and impure sealing using the same `>`/`>>` motif used to distinguish the total and partial forms of functors and functor signatures.

- I assume that all

  - *tyvar*'s appearing in a single sequence $(tyvar_1, \cdots, tyvar_n)$
  - *reclab*'s in a record expression, record pattern, or record type
  - *expid*'s bound in a single *funbinds*
  - *conid*'s and *expid*'s bound in a single *datbinds*

  are distinct.

9.2 Overview of Elaboration

9.2.1 Preliminaries

- I assume the existence of an injective overbar function $\overline{\cdot}$ mapping each EL identifier to an IL label. This function can also be applied pointwise to a "long" identifier, resulting in a sequence of labels (separated by dots), called *labs*. I will also assume the existence of an infinite set of "internal" labels that are not in the range of the overbar function. There is a fixed subset of the internal labels that the elaborator uses to recognize certain elaboration idioms. These known labels include "it", "in", "out", "pack", "unpack", "hidden", "visible", "tag", "fail", and the natural numbers. I will denote all other internal labels by *ilab*.

- A label *lab* may be "opened" by writing lab^*. Open labels are used to mark certain modules that are "open" for identifier lookup. (See the discussion of the lookup rules below for details.) One may also join two labels lab_1 and lab_2 to form a new, *internal* label $lab_1\_lab_2$. If a label is the result of joining, it may be deconstructed into its component parts. Joining of labels is useful in the elaboration of `datatype` and `packtype` bindings.

- The elaborator maintains a typing context Γ that is essentially a list of labeled declarations (*ldecs*), except for two generalizations: (1) the list may contain duplicate labels (but not duplicate variables), and (2) it may also contain declarations of the form $\overline{sigid}{=}sig$ for the purpose of elaborating EL `signature` bindings. The labels on the declarations in Γ are used to map EL identifiers to corresponding IL variables.

  I will write $\mathrm{IL}(\Gamma)$ to denote the erasure of the elaboration context Γ into an IL context *decs*, which simply strips off the labels on all the *ldec* entries in Γ, and erases all the signature declarations. I will take "$\vdash \Gamma$ ok" to mean that $\vdash \mathrm{IL}(\Gamma)$ ok and, for all signature declarations $\overline{sigid}{=}sig$ in Γ, we have $\mathrm{IL}(\Gamma) \vdash sig : \mathsf{Sig}$. For all other IL judgments $\mathcal{J}$, I will take "$\Gamma \vdash \mathcal{J}$" to mean that $\vdash \Gamma$ ok and $\mathrm{IL}(\Gamma) \vdash \mathcal{J}$.

- It is useful to work with a subclass of signatures, called *static signatures* and denoted by the metavariable *statsig*, whose distinguishing characteristic is that they do not contain any value or partial functor declarations. In essence, one may view a static signature as a *kind* that has been promoted to the status of a signature. Formally, static signatures and declarations are defined by the following grammar:

$$\begin{array}{rcl} statsig & ::= & [\![statldecs]\!] \\ & \mid & \Pi^{\mathsf{tot}}(mvar{:}statsig).statsig' \\ & \mid & \rho(statsig) \\ & \mid & \mathsf{maybe}(statsig) \end{array} \qquad \begin{array}{rcl} statldec & ::= & lab{\triangleright}statdec \\ statdec & ::= & cvar{:}knd \\ & \mid & mvar{:}statsig \end{array}$$

  Any signature can be erased into a static signature by the following $\mathsf{Stat}(\cdot)$ function:

$\mathsf{Stat}(sig)$	$\mathsf{Stat}([\![ldecs]\!])$	$= [\![\mathsf{Stat}(ldecs)]\!]$
	$\mathsf{Stat}(\Pi^{\mathsf{tot}}(mvar{:}sig).sig')$	$= \Pi^{\mathsf{tot}}(mvar{:}\mathsf{Stat}(sig)).\mathsf{Stat}(sig')$
	$\mathsf{Stat}(\Pi^{\mathsf{par}}(mvar{:}sig).sig')$	$= [\![\cdot]\!]$
	$\mathsf{Stat}(\rho(mvar).sig)$	$= \rho(\mathsf{Stat}(sig))$
	$\mathsf{Stat}(\mathsf{maybe}(sig))$	$= \mathsf{maybe}(\mathsf{Stat}(sig))$
$\mathsf{Stat}(ldecs)$	$\mathsf{Stat}(\cdot)$	$= \cdot$
	$\mathsf{Stat}(lab{\triangleright}cvar{:}knd, ldecs)$	$= lab{\triangleright}cvar{:}knd, \mathsf{Stat}(ldecs)$
	$\mathsf{Stat}(lab{\triangleright}evar{:}con, ldecs)$	$= \mathsf{Stat}(ldecs)$
	$\mathsf{Stat}(lab{\triangleright}mvar{:}sig, ldecs)$	$= lab{\triangleright}mvar{:}\mathsf{Stat}(sig), \mathsf{Stat}(ldecs)$

One can view the $\mathsf{Stat}(sig)$ function as doing half the work of the $\mathsf{Fst}(sig)$ function. It eliminates the "dynamic" part of *sig*, but does not go the extra step of turning the signature into a kind. During elaboration, $\mathsf{Stat}(sig)$ is often easier to work with than $\mathsf{Fst}(sig)$ because static signatures are amenable to certain operations (*e.g.,* looking up a label in a signature) that are not defined for kinds. Here are some useful properties of the $\mathsf{Stat}(\cdot)$ function:

1. If $cdecs \vdash sig : \mathsf{Sig}$, then $cdecs \vdash \mathsf{Stat}(sig) : \mathsf{Sig}$ and $cdecs \vdash \mathsf{Fst}(sig) \equiv \mathsf{Fst}(\mathsf{Stat}(sig))$.
2. If $cdecs \vdash sig_1 \equiv sig_2$, then $cdecs \vdash \mathsf{Stat}(sig_1) \equiv \mathsf{Stat}(sig_2)$.
3. If $cdecs \vdash sig_1 \leq sig_2$, then $cdecs \vdash \mathsf{Stat}(sig_1) \leq \mathsf{Stat}(sig_2)$.

- I write *vpmod* to denote a *pmod* that is also a *vmod*, *i.e.,* a module expression that is both projectible and valuable. I write *phrase* to denote something that is either a *con*, *exp* or *mod*. I write *class* to denote something that is either a *knd*, *con* or *sig*.

- As explained in Section 5.4, the first phase of recursive module elaboration generates something I call a *meta-signature*, which encapsulates the type information that is known at different points during the typechecking of the recursive module. Meta-signatures have the following grammar:

$$
\begin{array}{rcl}
metasig & ::= & [\![metaldecs]\!] \\
 & | & \Pi^{\mathsf{tot}}(mvar{:}sig).metasig' \\
 & | & \rho(mvar).metasig \\
 & | & \{\text{private}{=}sig_1, \text{public}{=}sig_2\}
\end{array}
\qquad
\begin{array}{rcl}
metaldec & ::= & lab{\triangleright}metadec \\
metadec & ::= & cvar{:}knd \\
 & | & evar{:}con \\
 & | & mvar{:}metasig
\end{array}
$$

The one case that distinguishes *metasig*'s from ordinary *sig*'s is $\{\text{private}{=}sig_1, \text{public}{=}sig_2\}$. I call this a "switchable" signature because it provides a way for the recursive module elaboration algorithm to switch between public and private views of a module when typechecking different parts of the recursive module body.

The functions $\mathsf{Priv}(\cdot)$ and $\mathsf{Pub}(\cdot)$ erase meta-signatures into ordinary signatures by switching all switchable signatures to either the private or the public setting, respectively:

$\mathsf{Priv}(metasig)$	$\mathsf{Priv}([\![metaldecs]\!])$	$= [\![\mathsf{Priv}(metaldecs)]\!]$
	$\mathsf{Priv}(\Pi^{\mathsf{tot}}(mvar{:}sig).metasig')$	$= \Pi^{\mathsf{tot}}(mvar{:}sig).\mathsf{Priv}(metasig')$
	$\mathsf{Priv}(\rho(mvar).metasig)$	$= \rho(mvar).\mathsf{Priv}(metasig)$
	$\mathsf{Priv}(\{\text{private}{=}sig_1, \text{public}{=}sig_2\})$	$= sig_1$
$\mathsf{Priv}(metaldecs)$	$\mathsf{Priv}(\cdot)$	$= \cdot$
	$\mathsf{Priv}(lab{\triangleright}cvar{:}knd, metaldecs)$	$= lab{\triangleright}cvar{:}knd, \mathsf{Priv}(metaldecs)$
	$\mathsf{Priv}(lab{\triangleright}evar{:}con, metaldecs)$	$= lab{\triangleright}evar{:}con, \mathsf{Priv}(metaldecs)$
	$\mathsf{Priv}(lab{\triangleright}mvar{:}metasig, metaldecs)$	$= lab{\triangleright}mvar{:}\mathsf{Priv}(metasig), \mathsf{Priv}(metaldecs)$
$\mathsf{Pub}(metasig)$	$\mathsf{Pub}([\![metaldecs]\!])$	$= [\![\mathsf{Pub}(metaldecs)]\!]$
	$\mathsf{Pub}(\Pi^{\mathsf{tot}}(mvar{:}sig).metasig')$	$= \Pi^{\mathsf{tot}}(mvar{:}sig).\mathsf{Pub}(metasig')$
	$\mathsf{Pub}(\rho(mvar).metasig)$	$= \rho(mvar).\mathsf{Pub}(metasig)$
	$\mathsf{Pub}(\{\text{private}{=}sig_1, \text{public}{=}sig_2\})$	$= sig_2$
$\mathsf{Pub}(metaldecs)$	$\mathsf{Pub}(\cdot)$	$= \cdot$
	$\mathsf{Pub}(lab{\triangleright}cvar{:}knd, metaldecs)$	$= lab{\triangleright}cvar{:}knd, \mathsf{Pub}(metaldecs)$
	$\mathsf{Pub}(lab{\triangleright}evar{:}con, metaldecs)$	$= lab{\triangleright}evar{:}con, \mathsf{Pub}(metaldecs)$
	$\mathsf{Pub}(lab{\triangleright}mvar{:}metasig, metaldecs)$	$= lab{\triangleright}mvar{:}\mathsf{Pub}(metasig), \mathsf{Pub}(metaldecs)$

In fact, $\mathsf{Priv}(metasig)$ and $\mathsf{Pub}(metasig)$ are just two possible settings of $metasig$, where a "setting" of $metasig$ is taken to mean an ordinary signature formed by replacing each switchable signature in $metasig$ of the form $\{\text{private}=sig_1, \text{public}=sig_2\}$ with either sig_1 or sig_2 (*i.e.,* setting each switch to "private" or "public"). Before observing some useful properties of meta-signatures and their settings, let us define what it means for a meta-signature to be well-formed by extending the existing IL judgments for signatures and declarations with the following new cases for meta-signatures and meta-declarations:

$$\boxed{cdecs \vdash metasig : \mathsf{Sig}}$$

$$\frac{cdecs \vdash metaldecs \text{ ok}}{cdecs \vdash [\![metaldecs]\!] : \mathsf{Sig}} \tag{9.1}$$

$$\frac{cdecs \vdash sig : \mathsf{Sig} \qquad cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(sig) \vdash metasig' : \mathsf{Sig}}{cdecs \vdash \Pi^{\mathsf{tot}}(mvar{:}sig).metasig' : \mathsf{Sig}} \tag{9.2}$$

$$\frac{\begin{array}{c} cdecs \vdash \mathsf{Fst}(\mathsf{Priv}(metasig)) : \mathsf{Kind} \\ cdecs, mvar^{\mathsf{c}}{:}\mathsf{Fst}(\mathsf{Pub}(metasig)) \vdash metasig : \mathsf{Sig} \end{array}}{cdecs \vdash \rho(mvar).metasig : \mathsf{Sig}} \tag{9.3}$$

$$\frac{cdecs \vdash sig_1 \leq sig_2}{cdecs \vdash \{\text{private}=sig_1, \text{public}=sig_2\} : \mathsf{Sig}} \tag{9.4}$$

$$\boxed{cdecs \vdash metaldecs \ \mathbf{ok}}$$

$$\frac{cdecs \vdash metadec \text{ ok} \qquad cdecs, \mathsf{Fst}(\mathsf{Pub}(metadec)) \vdash metaldecs \text{ ok}}{cdecs \vdash lab \triangleright metadec, metaldecs \text{ ok}} \tag{9.5}$$

$$\boxed{cdecs \vdash metadec \ \mathbf{ok}}$$

$$\frac{cdecs \vdash metasig : \mathsf{Sig}}{cdecs \vdash mvar{:}metasig \text{ ok}} \tag{9.6}$$

These well-formedness rules were designed in order to guarantee the following property:

– If $cdecs \vdash metasig : \mathsf{Sig}$ and sig is a setting of $metasig$, then $cdecs \vdash sig : \mathsf{Sig}$ and $cdecs \vdash \mathsf{Priv}(metasig) \leq sig$ and $cdecs \vdash sig \leq \mathsf{Pub}(metasig)$ (in particular, $cdecs \vdash \mathsf{Priv}(metasig) \leq \mathsf{Pub}(metasig)$).

While this property is provable by straightforward induction, its proof depends on some subtle points in the above rules. In Rule 9.2, it is important that the argument in the functor meta-signature is an ordinary signature, since functor subtyping is invariant in the argument. In Rule 9.3, the first premise is necessary to ensure that, for any setting sig of $metasig$, $\mathsf{Fst}(sig)$ does not refer to $mvar^{\mathsf{c}}$.

9.2.2 Guide to the Elaboration Judgments

In this section, I describe at a high level what each elaboration judgment means, how it is used, what its soundness properties are, and (in some cases) what its output may be expected to look like. Note: the soundness properties for all the rules assume that the given typing context Γ (or in some cases *decs*) is well-formed.

Main Translation Judgments

- $\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \mathbf{T}$
  - **Interpretation:** Given an EL type (*ty*), return its translation as an IL type (*con*).
  - **Soundness:** $\Gamma \vdash \mathit{con} : \mathbf{T}$.
- $\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con}$
  - **Interpretation:** Given an EL term (*expr*), return its translation as an IL term (*exp*) and its type (*con*).
  - **Soundness:** $\Gamma \vdash \mathit{exp} : \mathit{con}$.
- $\Gamma \vdash \mathit{match} \rightsquigarrow \mathit{val} : \mathit{con}$
  - **Interpretation:** Given an EL pattern match (*match*), return a function *val* (of type *con*) that takes an argument of the type expected by *match*, tries to match the argument against each of the *mrule*'s in the *match* (in order), and throws a "fail" exception if the pattern match fails.
  - **Soundness:** $\Gamma \vdash \mathit{val} : \mathit{con}$.
  - **Comments:** The "fail" exception is a component of the "basis" module that may be assumed bound in Γ (see Section 9.3.1 for details).
- $\Gamma \vdash \mathit{binding}(s) \rightsquigarrow \mathit{lbnds} :_\kappa \mathit{ldecs}$
  - **Interpretation:** Given EL *binding*(*s*), return their translation as a sequence of IL labeled bindings (*lbnds*), along with a matching sequence of labeled declarations (*ldecs*) that describe them. κ is the purity level of the *lbnds*.
  - **Soundness:** $\Gamma \vdash \mathit{lbnds} :_\kappa \mathit{ldecs}$.
  - **Comments:** The labels on the output *lbnds* fall into two general categories: (1) they have the form $\overline{\mathit{id}}$, where *id* is the EL identifier bound by the corresponding input *binding*, or (2) they have the form ilab^*, where *ilab* is an arbitrary internal label. In the latter case, the elaborator understands the binding to mean that the module labeled ilab^* should be considered "open." That is, the elaborator treats the components of the module as if they were bound at top level alongside the other category-1 *lbnds*.
- $\Gamma \vdash \mathit{modexp} \rightsquigarrow \mathit{mod} :_\kappa \mathit{sig}$
  - **Interpretation:** Given an EL module expression (*modexp*), return its translation as an IL module (*mod*), along with its principal signature (*sig*) and purity level κ.
  - **Soundness:** $\Gamma \vdash \mathit{mod} :_\kappa \mathit{sig}$.

- **Comments:** As explained in Section 4.2.6, I follow HS's approach to addressing the avoidance problem. For example, I elaborate the EL module `let` *bindings* `in` *modexp* `end` into the IL module $[\text{hidden}\triangleright mvar{=}[lbnds], \text{visible}^*{=}mod]$, where *lbnds* is the translation of *bindings* and *mod* is the translation of *modexp*. The "hidden" label ensures that the local *bindings* are in fact kept in a hidden namespace, because no EL identifier will be translated (by the overbar function) into the label "hidden". In contrast, since the visible* label on the second component is open, the components of *mod* will be accessible to the programmer. Several of the *modexp* translation rules return IL modules of this same form. I refer to such modules as having "existential" signature $\exists(mvar{:}sig_1).sig_2$, which is definable as $[\![\text{hidden}\triangleright mvar{:}sig_1, \text{visible}^*{:}sig_2]\!]$.

- $\Gamma \vdash spec(s) \rightsquigarrow ldecs$
  - **Interpretation:** Given EL $spec(s)$, return their translation as a sequence of IL *ldecs*.
  - **Soundness:** $\Gamma \vdash ldecs$ ok.
  - **Comments:** As in the translation of *bindings*, the output *ldecs* here fall into two categories: those with labels of the form $\overline{id}$ and those with labels of the form $ilab^*$. The interpretation of the latter category is the same as in the *binding* translation: the elaborator treats the components of the module declared with label $ilab^*$ as if they were declared in the same namespace as the other category-1 *ldecs*.

- $\Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}$
  - **Interpretation:** Given an EL signature expression (*sigexp*), return its translation as an IL signature (*sig*).
  - **Soundness:** $\Gamma \vdash sig : \mathsf{Sig}$.

- $\Gamma \vdash_{\overline{\mathsf{stat}}} spec(s) \rightsquigarrow statldecs$
  $\Gamma \vdash_{\overline{\mathsf{stat}}} sigexp \rightsquigarrow statsig : \mathsf{Sig}$
  - **Interpretation:** Same as the normal judgments for translating specifications and signatures, except that these judgments only translate the *static* part of the input (*i.e.,* `val` specs, `exception` specs, etc. are ignored).
  - **Soundness:** $\Gamma \vdash statldecs$ ok, $\Gamma \vdash statsig : \mathsf{Sig}$.
  - **Comments:** These judgments are useful in elaborating recursively dependent signatures. Given an EL rds of the form `rec (`*modid*`)` *sigexp*, we ultimately want to do full elaboration of *sigexp* into an IL signature *sig*. Before we can do full elaboration, however, we need to figure out what $\mathsf{Stat}(sig)$ is, because *sigexp* is only well-formed in a typing context where $\overline{modid}$ has signature $\mathsf{Stat}(sig)$. These static elaboration judgments allow us to compute $\mathsf{Stat}(sig)$ under the original context Γ first, without worrying about references to *modid* in the dynamic components of *sigexp*. Although soundness does not depend on it, these judgments are defined so that, whenever $\Gamma \vdash spec(s) \rightsquigarrow ldecs$ (resp. $\Gamma \vdash sigexp \rightsquigarrow sig : \mathsf{Sig}$), it is also the case that $\Gamma \vdash_{\overline{\mathsf{stat}}} spec(s) \rightsquigarrow \mathsf{Stat}(ldecs)$ (resp. $\Gamma \vdash_{\overline{\mathsf{stat}}} sigexp \rightsquigarrow \mathsf{Stat}(sig) : \mathsf{Sig}$).

- $\Gamma \vdash pat \Leftarrow exp : con \text{ else } val \rightsquigarrow lbnds : ldecs$
  - **Interpretation:** Given an EL pattern (*pat*), an IL expression *exp* of type *con*, and an exception *val* of type Tagged, match *exp* against the pattern *pat*. This results in a

bunch of IL bindings *lbnds* (described by *ldecs*), which bind pattern identifiers to the appropriate projections from *exp*. If the pattern match fails, the evaluation of *lbnds* will raise the exception *val*.

– **Soundness:** If $\Gamma \vdash \mathit{exp} : \mathit{con}$ and $\Gamma \vdash \mathit{val} : \mathsf{Tagged}$, then $\Gamma \vdash \mathit{lbnds} :_{\mathtt{P}} \mathit{ldecs}$. Furthermore, *lbnds* is a sequence of term bindings. No binding refers to any of the previous bindings in the sequence.

- $\Gamma \vdash \mathit{datbinds} \rightsquigarrow \mathit{sig}$
  - **Interpretation:** Given an EL `datatype` specification (*datbinds*), return an IL signature (*sig*) describing the specified types and their data constructors.
  - **Soundness:** $\Gamma \vdash \mathit{sig} : \mathsf{Sig}$.
  - **Comments:** Each type *conid* specified by the *datbinds* will be accorded its own (open) module declaration. In that module declaration will be specified: (1) the type itself (with an *opaque* kind, since `datatype`'s in ML are abstract types), (2) two coercion functions ($\overline{\mathit{conid}}$_in and $\overline{\mathit{conid}}$_out) for converting between $\overline{\mathit{conid}}$ and its recursive expansion, and (3) each of *conid*'s data constructors. Since `datatype` bindings/specifications are the only places where the “in” and “out” labels are used, a number of rules throughout the elaborator test whether a given type named *lab* is in fact a `datatype` by checking whether it is defined in the same module as a value labeled *lab*_in.

Canonical Implementations of Signatures

- $\mathit{decs} \vdash_{\mathrm{can}} \mathit{sig} \rightsquigarrow \mathit{mod}$
  - **Interpretation:** Given an IL signature *sig*, return a canonical module *mod* of that signature.
  - **Soundness:** $\mathit{decs} \vdash \mathit{mod} :_{\mathtt{P}} \mathit{sig}$.
  - **Comments:** Canonical implementations exist only for signatures of a certain restricted form. Essentially, the input signature must be static and transparent—as, in general, neither value specifications nor opaque type specifications have canonical implementations—the only exception being that it may contain module declarations that correspond to the elaboration of `datatype` specifications. This judgment is invoked by Rule 9.152 for recursive module elaboration and also by Rule 9.41 for translating `datatype` bindings.

Coercive Signature Matching

- $\mathit{decs} \vdash_{\mathrm{sub}} \mathit{vpmod} : \mathit{sig}_0 \preceq \mathit{sig} \rightsquigarrow \mathit{pmod} : \mathit{tsig}$
  - **Interpretation:** Given a projectible, valuable module *vpmod* of signature sig_0, and a target signature *sig*, coerce *vpmod* into *sig*. The coercion module *pmod* (with transparent signature *tsig*) will copy all of its components from *vpmod* but have the shape of *sig* instead of sig_0.
  - **Soundness:** If $\mathit{decs} \vdash \mathit{vpmod} :_{\mathtt{P}} \mathit{sig}_0$ and $\mathit{decs} \vdash \mathit{sig} : \mathsf{Sig}$, then $\mathit{decs} \vdash \mathit{pmod} :_{\mathtt{P}} \mathit{tsig}$ and $\mathit{decs} \vdash \mathit{tsig} \leq \mathit{sig}$.
  - **Comments:** Unlike IL subtyping, coercive signature matching permits dropping and reordering of structure components, allows total functors to be coerced to partial functor

signatures, and matches functor signatures contravariantly in the argument and covariantly in the result.

- $decs \vdash_{\mathrm{sub}} vpmod : sig_0 \preceq ldec(s) \rightsquigarrow plbnd(s) : tldec(s)$
  - **Interpretation:** Given a projectible, valuable module *vpmod* of signature sig_0, and target specification(s) $ldec(s)$, return $plbnd(s)$ that copy the corresponding components from *vpmod* and match the shape of the $ldec(s)$.
  - **Soundness:** If $decs \vdash vpmod :_{\mathrm{P}} sig_0$ and $decs \vdash ldec(s)$ ok, then $decs \vdash plbnd(s) :_{\mathrm{P}} tldec(s)$ and $decs \vdash tldec(s) \leq ldec(s)$.

Signature Patching

- $sig \vdash_{\mathrm{wt}} labs := phrase \rightsquigarrow sig'$
  - **Interpretation:** By adding to the signature *sig* the fact that the type or module component selected by *labs* is statically equivalent to *phrase*, we get the signature sig'.
  - **Comments:** This judgment is used in the translation of signatures with `where type` constraints (Rule 9.83). It does not guarantee that the output sig' is well-formed or that sig' is a subtype of *sig*—Rule 9.83 verifies this independently. The judgment simply traverses *sig* according to the path described by *labs* and then replaces whatever classifier (*class*) it finds there by $\mathfrak{S}(phrase : class)$.

- $sig \vdash_{\mathrm{sh}} labs_1 := labs_2 \rightsquigarrow sig'$
  - **Interpretation:** By adding to the signature *sig* the fact that the type or module component selected by $labs_1$ is statically equivalent to the one selected by $labs_2$, we get the signature sig'.
  - **Comments:** This judgment is used in the translation of type sharing and structure sharing constraints (Rules 9.73 and 9.74). Like the $\vdash_{\mathrm{wt}}$ judgment, it does not guarantee that the output sig' is well-formed or that it is a subtype of *sig*. The judgment simply traverses *sig* according to the path described by the common prefix of $labs_1$ and $labs_2$, and then it invokes the $\vdash_{\mathrm{wt}}$ judgment to finish the job. It only succeeds if the component indexed by $labs_1$ appears further down in the input *sig* than the one indexed by $labs_2$ (otherwise, the classifier of the $labs_1$ component could not be replaced by a singleton referring to the $labs_2$ component).

A note about type and structure sharing: The semantics of `sharing type` constraints given by the Definition of SML makes a subtle distinction between "flexible" and "rigid" type components. Given a signature *sig*, a flexible type is one that *sig* specifies opaquely or that it specifies as transparently equal to some other type that *sig* specifies opaquely. A type is rigid if it is not flexible. The Definition only permits `sharing` of two flexible types. Thus, the signature

```
sig type t = int type u = int sharing type t = u end
```

is not considered well-formed, because `t` and `u` are transparently specified to equal a type (`int`) not specified by the signature and are therefore rigid.

The Definition's restriction to sharing flexible types was put in place to prevent type sharing from requiring higher-order unification. While the restriction is not a problem *per se*, it becomes problematic because of the way the Definition defines *structure sharing* constraints in terms of type sharing constraints. Specifically, the Definition defines a sharing constraint on two structure components of a signature by expanding it into a set of type sharing constraints, one for every type constructor of the same name and arity provided by both structures. As a result, even if two structures are specified with exactly the same signature, attempting to impose a sharing constraint on them will fail if that signature includes *even one* rigid type specification. Since, in my experience structure sharing is desired almost exclusively for sharing two structures of the same signature, I find the Definition's semantics of structure sharing to be overly restrictive.

Different implementations of SML loosen the Definition in different ways. The SML/NJ compiler allows any two structure components to be shared if they are specified by the same signature *identifier*. The TILT compiler employs a more liberal semantics of type sharing so that one may share two rigid type components so long as they are transparently equal to the same type.

The approach I have taken in my present design is to dispense with the rigid/flexible distinction entirely. I view a type/structure sharing constraint as simply a symmetric way of asking for one of the components to be assigned a singleton kind/signature referring to the other. The constraint is only considered valid if the resulting signature is an IL subtype of the original signature. If the types/structures did not have the same shape to begin with, then the sharing constraint may fail under my semantics but not under the Definition's. However, as the vast majority of sharing constraints are imposed on types/structures of exactly the same kind/signature, my semantics will in most realistic cases be more liberal than the Definition's.

Signature Peeling

- $\mathit{decs} \vdash_{\text{peel}} \mathit{pmod}{:}\mathit{sig} \rightsquigarrow \mathit{pmod}' : \mathit{sig}'$
  - **Interpretation:** Given a module *pmod* of signature *sig*, peel off the outer layers of *sig* until we reach an "interesting" signature, *i.e.,* either a functor or structure signature sig' that is not an existential signature (not of the form $[\![\text{hidden}{\triangleright}\mathit{mvar}{:}\mathit{sig}_1, \text{visible}^*{:}\mathit{sig}_2]\!]$). Correspondingly, pmod' is formed from *pmod* by a sequence of visible* projections, unroll'ings, and fetch'es.
  - **Soundness:** $\mathit{decs} \vdash \mathit{pmod}' :_{\mathrm{P}} \mathit{sig}'$.
  - **Comments:** This judgment is useful when we want to check, for example, that a given module is really a functor, but we do not have a particular target signature that we are matching it against (Rule 9.49). It also inserts automatically the fetch'es of recursive module variables and unroll'ings of rds's that the EL syntax leaves implicit. I will say that a module *pmod* is in "peeled form" if $\mathit{decs} \vdash_{\text{peel}} \mathit{pmod}{:}\mathit{sig} \rightsquigarrow \mathit{pmod} : \mathit{sig}$.

Label Lookup

- $\Gamma \vdash_{\text{ctx}} \mathit{labs} \rightsquigarrow \mathit{phrase} : \mathit{class}$
  - **Interpretation:** Lookup the component indexed by *labs* in the context Γ and return it (*phrase*) along with a *class* describing it.
  - **Soundness:** $\Gamma \vdash \mathit{phrase} : \mathit{class}$. If *phrase* is a module, it is a *pmod* and it is guaranteed to be in peeled form, *i.e.,* $\Gamma \vdash_{\text{peel}} \mathit{phrase}{:}\mathit{class} \rightsquigarrow \mathit{phrase} : \mathit{class}$.
  - **Comments:** The lookup algorithm looks for the first *lab* in *labs*. It searches right-to-left through Γ, and looks inside the signatures of modules that have open labels (ilab^*) using

the signature lookup judgment (below). Once it finds the first *lab*, it uses the signature lookup judgment to look for the remaining *labs* if there are any.

- $decs; pmod{:}sig \vdash_{\mathrm{sig}} labs \rightsquigarrow phrase : class$
  - **Interpretation:** Given a module *pmod* of signature *sig*, lookup the component indexed by *labs* in *sig*, and return the resulting *phrase* (and its *class*), which will be a projection from *pmod*.
  - **Soundness:** $\Gamma \vdash phrase : class$. If *phrase* is a module, it is a *pmod* and it is guaranteed to be in peeled form, *i.e.,* $\Gamma \vdash_{\mathrm{peel}} phrase{:}class \rightsquigarrow phrase : class$.
  - **Comments:** This judgment is defined in a similar manner to the context lookup judgment—it searches for the head of *labs* right-to-left among the specifications that comprise *sig*, and then calls itself recursively if there are more *labs*.

Recursive Module Elaboration

- $\Gamma \vdash_{\mathrm{stat}} binding(s) \rightsquigarrow metaldec(s)$
  $\Gamma \vdash_{\mathrm{stat}} modexp \rightsquigarrow metasig$
  - **Interpretation:** Given an EL *binding(s)* or *modexp*, perform static elaboration of it (ignoring `val` bindings, etc.) and return a meta-signature or meta-declarations that reflect what type information is known in different parts of the input.
  - **Soundness:** $\Gamma \vdash metaldec(s)$ ok, $\Gamma \vdash metasig : \mathsf{Sig}$.
  - **Comments:** This is the first phase of recursive module elaboration, and it only succeeds if the input is pure/separable, *i.e.,* has purity classifier `P`. See Section 5.4 for a high-level explanation. This judgment is also used by Rule 9.13 in order to determine whether a module expression is projectible. Specifically, if the input *modexp* is projectible, then it will not contain any `datatype` bindings or any uses of sealing, so the output *metasig* will have the form of a normal signature (in fact, it will be a transparent, static signature).

- $\Gamma; metadec; \Gamma' \vdash_{\mathrm{rec}} pmod \Rightarrow binding(s) \rightsquigarrow lbnd(s) : tldec(s)$
  $\Gamma; metadec; \Gamma' \vdash_{\mathrm{rec}} pmod \Rightarrow modexp \rightsquigarrow mod : tsig$
  - **Interpretation:** The main phase of recursive module elaboration.
  - **Soundness:** Assuming $\Gamma \vdash metadec$ ok and $\vdash \Gamma, \mathsf{Pub}(metadec), \Gamma'$ ok,
    then $\Gamma, \mathsf{Priv}(metadec), \Gamma' \vdash lbnd(s) :_{\mathtt{P}} tldec(s)$ and $\Gamma, \mathsf{Priv}(metadec), \Gamma' \vdash mod :_{\mathtt{P}} tsig$.
  - **Comments:** *pmod* tells us what part of the recursive module body we are currently elaborating, and the *metadec* in the context allows us to switch from public to private knowledge of certain type information when elaboration goes underneath a sealed module expression. Note that the soundness property is rather weak: it only says that the output is well-formed assuming the full private setting of the *metadec*. This is because different parts of the input are typechecked under different settings of *metadec*, and $\mathsf{Priv}(metadec)$ is the least common denominator in the subtyping hierarchy. For more details, see Section 9.3.8.

9.3 Elaboration

9.3.1 A Few More Preliminaries

- Given an elaboration context Γ, let $\mathrm{labdom}(\Gamma)$ be the set of labels on the entries in Γ, and let $\mathrm{vardom}(\Gamma)$ be $\mathrm{dom}(\mathrm{IL}(\Gamma))$.

- I will sometimes write $\mathit{var}{=}\mathit{phrase}$ or $\mathit{lab}{=}\mathit{phrase}$ instead of $\mathit{lab}{\triangleright}\mathit{var}{=}\mathit{phrase}$ when I do not care what *lab* or *var* is, respectively. In either case, to form a proper *lbnd*, one can choose some fresh internal label or variable to fill in whatever was omitted. Similarly, I will sometimes write $\mathit{var}{:}\mathit{class}$ or $\mathit{lab}{:}\mathit{class}$ instead of $\mathit{lab}{\triangleright}\mathit{var}{:}\mathit{class}$ when I do not care what *lab* or *var* is, respectively. In either case, to form a proper *ldec*, one can choose some fresh internal label or variable to fill in whatever was omitted.

- I will sometimes extend an elaboration context Γ with a *metaldec*. This is shorthand for extending Γ with $\mathsf{Pub}(\mathit{metaldec})$.

- Given a list of *ldecs*, $\mathrm{labdom}(\mathit{ldecs})$ is not an entirely accurate description of the labels that *ldecs* exports, since it does not take into account labels of components in the signatures of open modules. The function $\mathrm{vislabs}(\mathit{ldecs})$, defined here, computes more accurately the set of "visible" labels, *i.e.,* the labels that the signature lookup routine will succeed in finding if it looks for them in the signature $[\![\mathit{ldecs}]\!]$. This function comes in handy in Rule 9.57.

$\mathrm{vislabs}(\mathit{ldecs})$	$\mathrm{vislabs}(\cdot)$ $\mathrm{vislabs}(\mathit{ldec}, \mathit{ldecs})$	$= \emptyset$ $= \mathrm{vislabs}(\mathit{ldec}) \cup \mathrm{vislabs}(\mathit{ldecs})$
$\mathrm{vislabs}(\mathit{ldec})$	$\mathrm{vislabs}(\mathit{lab}{:}\mathit{knd})$ $\mathrm{vislabs}(\mathit{lab}{:}\mathit{con})$ $\mathrm{vislabs}(\mathit{lab}{:}\mathit{sig})$ $\mathrm{vislabs}(\mathit{lab}^*{:}[\![\mathit{ldecs}]\!])$ $\mathrm{vislabs}(\mathit{lab}^*{:}\rho(\mathit{mvar}).\mathit{sig})$	$= \{\mathit{lab}\}$ $= \{\mathit{lab}\}$ $= \{\mathit{lab}\}$, if *lab* is not open $= \mathrm{vislabs}(\mathit{ldecs})$ $= \mathrm{vislabs}(\mathit{lab}^*{:}\mathit{sig})$

- Following HS, I handle shadowing of EL identifiers by means of an operation of *syntactic concatenation with renaming*. Written $(\mathit{lbnds}{+}{+}\mathit{lbnds}') : (\mathit{ldecs}{+}{+}\mathit{ldecs}')$, this operation takes two lists of *lbnds* (and corresponding *ldecs*) and joins them together, taking care to correct for duplicate labels by relabeling any shadowed *lbnds*/*ldecs* of the first list with fresh internal labels. Defined as follows, it comes in handy in Rule 9.30:

$$
\begin{array}{l}
(\cdot\,{+}{+}\mathit{lbnds}') : (\cdot\,{+}{+}\mathit{ldecs}') \stackrel{\text{def}}{=} \mathit{lbnds}' : \mathit{ldecs}' \\
(\mathit{lab}{\triangleright}\mathit{bnd}, \mathit{lbnds}{+}{+}\mathit{lbnds}') : (\mathit{lab}{\triangleright}\mathit{dec}, \mathit{ldecs}{+}{+}\mathit{ldecs}') \stackrel{\text{def}}{=} \\
\quad \begin{cases}
\mathit{lab}{\triangleright}\mathit{bnd}, \mathit{lbnds}'' : \mathit{lab}{\triangleright}\mathit{dec}, \mathit{ldecs}'' & \text{if } \mathit{lab} \notin \mathrm{labdom}(\mathit{ldecs}'') \\
\mathit{ilab}{\triangleright}\mathit{bnd}, \mathit{lbnds}'' : \mathit{ilab}{\triangleright}\mathit{dec}, \mathit{ldecs}'' & \text{otherwise, where } \mathit{ilab} \notin \mathrm{labdom}(\mathit{ldecs}'') \\
& \text{and } \mathit{ilab} \text{ is open iff } \mathit{lab} \text{ is}
\end{cases} \\
\quad \text{where } \mathit{lbnds}'' : \mathit{ldecs}'' = (\mathit{lbnds}{+}{+}\mathit{lbnds}') : (\mathit{ldecs}{+}{+}\mathit{ldecs}')
\end{array}
$$

- The elaborator assumes the presence of a structure $\mathit{basis}{:}[\![\mathit{ldecs}_{\mathit{basis}}]\!]$ serving as the initial basis for the internal language. $\mathit{ldecs}_{\mathit{basis}}$ must contain at least the following fields, which define three exceptions, as well as equality functions $\mathsf{eq}_{\mathit{con}}$ for some fixed set of base types *con*:

$$
\begin{array}{l}
\overline{\texttt{Bind}}^* \;{:}[\![\mathrm{tag}{:}\mathsf{Unit\ Tag}, \overline{\texttt{Bind}}{:}\mathsf{Tagged}]\!], \\
\overline{\texttt{Match}}^*{:}[\![\mathrm{tag}{:}\mathsf{Unit\ Tag}, \overline{\texttt{Match}}{:}\mathsf{Tagged}]\!], \\
\mathrm{fail}^* \quad{:}[\![\mathrm{tag}{:}\mathsf{Unit\ Tag}, \mathrm{fail}{:}\mathsf{Tagged}]\!]
\end{array}
$$

- In a number of rules, certain premises have the form *metavariable* = *something*, whereas others have the form *metavariable* := *something*. I take "=" and ":=" both to mean syntactic equality. However, in order to read the rules algorithmically, one should interpret the former as "has the form", and the latter as "is defined to be".

- In a number of rules, I write a wildcard _ in the output of a premise when it is important that the premise succeed, but it is irrelevant what the output is.

- Optional elements in rules are enclosed in angle brackets. For each rule, either all or none of the elements in angle brackets must be present. In some cases, the optional element notation is insufficient. Therefore, we have the additional notation

$$\left\{\begin{array}{c} \mathit{element_1} \\ \text{or} \\ \mathit{element_2} \end{array}\right\}$$

  which means that either *element₁* or *element₂* must be present. If there are multiple such choices in a single rule, this means that either the first element should be chosen in all cases, or the second element should be chosen in all cases. An extension of this notation gives the choices subscripts. Then all choices with the same subscript must agree (all first element or all second element) but two choices with different subscripts are completely independent.

- To make things more concise, I often view the monomorphic instance of an expression as a special case of the polymorphic instance. Except when otherwise noted, the kind $\mathbf{T}^n \rightarrow \mathbf{T}$, the type $\forall(\mathit{cvar}_p{:}\mathit{knd}_p).\mathit{con}$ and the term $\mathit{exp}[\mathit{con}_p]$ may be considered shorthand for $\mathbf{T}$, *con* and *exp*, respectively, in the monomorphic instance when $n = 0$.

- There are some kinds that may also be viewed as signatures, *e.g.*, $\mathit{knd} = [\![1{:}\mathbf{T}, \cdots, n{:}\mathbf{T}]\!]$. It should be clear from context which syntactic class is intended. For instance, if *knd* is used to classify a *cvar*, then it is a kind; but if *knd* is used to classify a *mvar*, then it is a signature.

As shown in Figure 9.3, the elaborator makes use of a number of derived forms of IL objects, most of which are self-explanatory. A few points of note:

- A number of the derived form definitions invent new variables on the right-hand side. These variables are always assumed to be fresh in the sense that they do not clash with the free variables of the expression.

- I will sometimes write λ-expressions without the result type when it can be inferred from context.

- plet is a projectible encoding of module-level let, assuming the submodules are projectible. elet stands for "existential" let, and is used to introduce modules of "existential" signature. See the translation of module expressions for examples of how these derived forms are used.

- $\mathsf{catch}^{\mathit{con}}\ \mathit{exp}\ \mathsf{with}\ \mathit{exp}'$ attempts to evaluate *exp* (which has type *con*). If an exception is raised, catch checks whether it is the "fail" exception. If so, exp' is evaluated; if not, the exception is reraised.

- $\mathsf{pproj}_{\mathit{lab}_k}^{\Sigma\{(\mathit{lab}_i \mapsto \mathit{con}_i)_{i=1}^n\}}(\mathit{exp}, \mathit{exp}')$ evaluates *exp* to a value of the given sum type, and then checks whether it is tagged with lab_k. If so, it projects out the underlying value (of type con_k); if not, the exception exp' is raised.

$$
\begin{array}{lcl}
\mathsf{Unit} & \stackrel{\text{def}}{=} & \{\} \\
\mathsf{Bool} & \stackrel{\text{def}}{=} & \Sigma\{1{:}\mathsf{Unit}, 2{:}\mathsf{Unit}\} \\
\mathsf{false} & \stackrel{\text{def}}{=} & \mathsf{inj}_1^{\mathsf{Bool}}\,\{\} \\
\mathsf{true} & \stackrel{\text{def}}{=} & \mathsf{inj}_2^{\mathsf{Bool}}\,\{\} \\
\mathsf{if}\ exp_0\ \mathsf{then}\ exp_2\ \mathsf{else}\ exp_1 & \stackrel{\text{def}}{=} & \mathsf{case}\ exp_0\ \mathsf{of}\ (i \mapsto \lambda evar{:}\mathsf{Unit}.exp_i)_{i=1}^2\ \mathsf{end} \\
\lambda(evar{:}con){:}con'.exp & \stackrel{\text{def}}{=} & \mathsf{fix}_1\ evar'(evar{:}con){:}con'{=}exp\ \mathsf{end} \\
\\
knd^n & \stackrel{\text{def}}{=} & \{1{:}knd, \cdots, n{:}knd\} \\
knd_1 \times \cdots \times knd_n & \stackrel{\text{def}}{=} & \{1{:}knd_1, \cdots, n{:}knd_n\} \\
(con_1, \cdots, con_n) & \stackrel{\text{def}}{=} & \{1{=}con_1, \cdots, n{=}con_n\} \\
con_1 \times \cdots \times con_n & \stackrel{\text{def}}{=} & \{\overline{1}{:}con_1, \cdots, \overline{n}{:}con_n\} \\
(exp_1, \cdots, exp_n) & \stackrel{\text{def}}{=} & \{\overline{1}{=}exp_1, \cdots, \overline{n}{=}exp_n\} \\
\Pi(cvar_1, \cdots, cvar_n).knd & \stackrel{\text{def}}{=} & \Pi(cvar{:}\mathbf{T}^n).knd[cvar.i/cvar_i]_{i=1}^n \\
\lambda(cvar_1, \cdots, cvar_n).con & \stackrel{\text{def}}{=} & \lambda(cvar{:}\mathbf{T}^n).con[cvar.i/cvar_i]_{i=1}^n \\
\forall(cvar_1, \cdots, cvar_n).con & \stackrel{\text{def}}{=} & \forall(cvar{:}\mathbf{T}^n).con[cvar.i/cvar_i]_{i=1}^n \\
\Lambda(cvar_1, \cdots, cvar_n).exp & \stackrel{\text{def}}{=} & \Lambda(cvar{:}\mathbf{T}^n).exp[cvar.i/cvar_i]_{i=1}^n \\
\\
\mathsf{let}\ cvar{=}con\ \mathsf{in}\ (mod : sig) & \stackrel{\text{def}}{=} & \mathsf{let}\ mvar{=}[1{=}con]\ \mathsf{in}\ (mod[mvar^{\mathsf{c}}.1/cvar] : sig) \\
(mod.lab : con) & \stackrel{\text{def}}{=} & \mathsf{let}\ mvar{=}mod\ \mathsf{in}\ (mvar.lab : con),\ \text{when } mod \neq pmod \\
(mod.lab : sig) & \stackrel{\text{def}}{=} & \mathsf{let}\ mvar{=}mod\ \mathsf{in}\ (mvar.lab : sig),\ \text{when } mod \neq pmod \\
\mathsf{plet}\ mvar{=}pmod_1\ \mathsf{in}\ pmod_2 & \stackrel{\text{def}}{=} & [1{\triangleright}mvar{=}pmod_1, 2{=}pmod_2].2 \\
\mathsf{elet}\ mvar{=}mod_1\ \mathsf{in}\ mod_2 & \stackrel{\text{def}}{=} & [\text{hidden}{\triangleright}mvar{=}mod_1, \text{visible}^*{=}mod_2] \\
\exists(mvar{:}sig_1).sig_2 & \stackrel{\text{def}}{=} & [\![\text{hidden}{\triangleright}mvar{:}sig_1, \text{visible}^*{:}sig_2]\!] \\
\\
\mathsf{fail}^{con} & \stackrel{\text{def}}{=} & \mathsf{raise}^{con}\ basis.\text{fail}^*.\text{fail} \\
\mathsf{catch}^{con}\ exp\ \mathsf{with}\ exp' & \stackrel{\text{def}}{=} & \mathsf{handle}\ exp\ \mathsf{with}\ \lambda evar{:}\mathsf{Tagged}. \\
 & & \quad \mathsf{iftagof}\ evar\ \mathsf{is}\ basis.\text{fail}^*.\text{tag} \\
 & & \quad \mathsf{then}\ \lambda evar{:}\mathsf{Unit}.exp'\ \mathsf{else}\ \mathsf{raise}^{con}\ evar \\
\mathsf{pproj}_{lab_k}^{\Sigma\{(lab_i \mapsto con_i)_{i=1}^n\}}\ (exp, exp') & \stackrel{\text{def}}{=} & \mathsf{case}\ exp\ \mathsf{of} \\
 & & \ lab_1 \mapsto \lambda evar{:}con_1.\mathsf{raise}^{con_k}\ exp', \cdots, \\
 & & \ lab_k \mapsto \lambda evar{:}con_k.evar, \cdots, \\
 & & \ lab_n \mapsto \lambda evar{:}con_n.\mathsf{raise}^{con_k}\ exp'\ \mathsf{end}
\end{array}
$$

Figure 9.3: Derived Forms

- Although I have not written their definitions down here, I assume that less restrictive versions of most IL term constructs have been defined (in terms of let) that permit subterms to be arbitrary terms, not just values, and that evaluate those subterms in left-to-right order. For example, I treat $exp_1\ exp_2$ as shorthand for $\mathsf{let}\ evar_1{=}exp_1\ \mathsf{in}\ \mathsf{let}\ evar_2{=}exp_2\ \mathsf{in}\ evar_1\ evar_2$.

9.3.2 Main Translation Rules

Type Expressions $\boxed{\Gamma \vdash ty \rightsquigarrow con : \mathbf{T}}$

$$\frac{}{\Gamma \vdash \texttt{int} \rightsquigarrow \mathsf{Int} : \mathbf{T}} \tag{9.7}$$

Rule 9.7: There are analogous rules for the other base types (`int`, `bool`, `unit`, `string`, etc.), and the base type constructors (`ref`).

$$\frac{}{\Gamma \vdash \texttt{exn} \rightsquigarrow \mathsf{Tagged} : \mathbf{T}} \tag{9.8}$$

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{tyvar} \rightsquigarrow con : \mathbf{T}}{\Gamma \vdash tyvar \rightsquigarrow con : \mathbf{T}} \tag{9.9}$$

Rule 9.9: As ML does not support higher-kinded polymorphism at the core language level, type variables all have kind $\mathbf{T}$.

$$\frac{\forall i \in 1..n : \quad \Gamma \vdash ty_i \rightsquigarrow con_i : \mathbf{T}}{\Gamma \vdash \texttt{\{}reclab_1 \texttt{:} ty_1 \texttt{,} \cdots \texttt{,} reclab_n \texttt{:} ty_n\texttt{\}} \rightsquigarrow \{\overline{reclab_1}{:}con_1, \cdots, \overline{reclab_n}{:}con_n\} : \mathbf{T}} \tag{9.10}$$

$$\frac{\Gamma \vdash ty \rightsquigarrow con : \mathbf{T} \qquad \Gamma \vdash ty' \rightsquigarrow con' : \mathbf{T}}{\Gamma \vdash ty \texttt{ -> } ty' \rightsquigarrow con \rightarrow con' : \mathbf{T}} \tag{9.11}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{longconid} \rightsquigarrow con : \mathbf{T}^n \rightarrow \mathbf{T} \\ \forall i \in 1..n : \quad \Gamma \vdash ty_i \rightsquigarrow con_i : \mathbf{T}\end{array}}{\Gamma \vdash \texttt{(}ty_1 \texttt{,} \cdots \texttt{,} ty_n\texttt{)}\ longconid \rightsquigarrow con\,(con_1, \cdots, con_n) : \mathbf{T}} \tag{9.12}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{stat}} modexp \rightsquigarrow tstatsig \\ \Gamma, mvar{:}tstatsig; mvar{:}tstatsig \vdash_{\text{sig}} \overline{conid} \rightsquigarrow con' : \mathbf{T}^n \rightarrow \mathbf{T} \\ con := con'[\mathsf{Can}(\mathsf{Fst}(tstatsig))/mvar^{\mathsf{c}}] \\ \forall i \in 1..n : \quad \Gamma \vdash ty_i \rightsquigarrow con_i : \mathbf{T} \\ modexp \text{ is not of the form } longmodid\end{array}}{\Gamma \vdash \texttt{(}ty_1 \texttt{,} \cdots \texttt{,} ty_n\texttt{)}\ modexp.conid \rightsquigarrow con(con_1, \cdots, con_n) : \mathbf{T}} \tag{9.13}$$

Rule 9.13: Projecting a type from a module. We must check that *modexp* is projectible. To do this, we perform static elaboration (aka the first phase of recursive module elaboration) and check that the static signature of *modexp* is in fact transparent. (Recall that transparency and projectibility amount to the same thing.) We can then project out the *conid* component of $\mathsf{Can}(\mathsf{Fst}(tstatsig))$.

The reason we perform static elaboration on *modexp* instead of ordinary elaboration is that the dynamic components of *modexp* are irrelevant as far as projection of its type components is concerned. Just as $\mathsf{Fst}(mod)$ may be well-formed even if *mod* is not, so *modexp.conid* may be well-formed even if *modexp* is not.

Term Expressions

$$\boxed{\Gamma \vdash expr \leadsto exp : con}$$

$$\frac{}{\Gamma \vdash scon \leadsto scon : \mathrm{type}(scon)} \tag{9.14}$$

Rule 9.14: We assume a meta-level function type which gives the IL type of each constant.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{longexpid} \leadsto exp : \forall(cvar_p{:}knd_p).con \\ \Gamma \vdash con_p : knd_p\end{array}}{\Gamma \vdash longexpid \leadsto exp[con_p] : con[con_p/cvar_p]} \tag{9.15}$$

Rule 9.15: Potentially polymorphic variables. This is an instance of nondeterminism, where we must guess the con_p with which to instantiate the type arguments of exp, if there are any.

$$\frac{\begin{array}{c}\Gamma \vdash modexp \leadsto mod :_\kappa sig \\ \Gamma, mvar{:}sig; mvar{:}sig \vdash_{\text{sig}} expid \leadsto exp : \forall(cvar_p{:}knd_p).con' \\ \Gamma, mvar{:}sig, cvar_p{:}knd_p \vdash con' \equiv con : \mathbf{T} \\ \Gamma, cvar_p{:}knd_p \vdash con : \mathbf{T} \qquad \Gamma \vdash con_p : knd_p \\ modexp \text{ is not of the form } longmodid\end{array}}{\Gamma \vdash modexp.expid \leadsto \mathsf{let}\ mvar{=}mod\ \mathsf{in}\ (exp[con_p] : con[con_p/cvar_p]) : con[con_p/cvar_p]} \tag{9.16}$$

Rule 9.16: $modexp.expid$ is elaborated as if it were `let module` $modid = modexp$ `in` $modid.expid$ `end`. See the elaboration of `let` bindings (Rule 9.18 below).

$$\frac{\forall i \in 1..n : \quad \Gamma \vdash expr_i \leadsto exp_i : con_i}{\begin{array}{l}\Gamma \vdash \texttt{\{}reclab_1 \texttt{ = } expr_1\texttt{,} \cdots \texttt{,}\ reclab_n \texttt{ = } expr_n\texttt{\}} \leadsto \\ \qquad \{\overline{reclab_1}{=}exp_1, \cdots, \overline{reclab_n}{=}exp_n\} : \{\overline{reclab_1}{:}con_1, \cdots, \overline{reclab_n}{:}con_n\}\end{array}} \tag{9.17}$$

$$\frac{\begin{array}{c}\Gamma \vdash bindings \leadsto lbnds : ldecs \\ \Gamma, ilab^*{\triangleright}mvar{:}[\![ldecs]\!] \vdash expr \leadsto exp : con' \\ \Gamma, mvar{:}[\![ldecs]\!] \vdash con' \equiv con : \mathbf{T} \qquad \Gamma \vdash con : \mathbf{T}\end{array}}{\Gamma \vdash \texttt{let}\ bindings\ \texttt{in}\ expr\ \texttt{end} \leadsto \mathsf{let}\ mvar{=}[lbnds]\ \mathsf{in}\ (exp : con) : con} \tag{9.18}$$

Rule 9.18: The "open label" convention is used here to make the local *bindings* accessible while translating *expr*. The elaborator verifies that the translated expression can be given a a type, which must not depend on any abstract types defined by *bindings*. As a practical matter, this can always be achieved by computing the normal form of con' (by Stone and Harper's normalization algorithm [75]) and then checking to make sure that $mvar^{\mathsf{c}}$ is not free in it.

$$\frac{\begin{array}{c}\Gamma \vdash expr \leadsto exp : con'' \to con \\ \Gamma \vdash expr' \leadsto exp' : con' \\ \Gamma \vdash con' \equiv con'' : \mathbf{T} \\ \text{Rule 9.20 does not apply.}\end{array}}{\Gamma \vdash expr\ \ expr' \leadsto exp\ exp' : con} \tag{9.19}$$

Rule 9.19: General function application, where exp is not a `datatype` constructor.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \textit{longexpid} \leadsto \textit{pmod}.\textit{lab}_i : \forall(\textit{cvar}_p{:}\textit{knd}_p).\textit{con}_i \to \textit{con} \\ \Gamma \vdash \textit{pmod}.\overline{\textit{conid}}\_\text{in} : \forall(\textit{cvar}_p{:}\textit{knd}_p).\textit{con}^{\text{sum}} \leadsto \textit{con} \\ \textit{con}^{\text{sum}} = \Sigma\{\textit{lab}_1{:}\textit{con}_1, \ldots, \textit{lab}_n{:}\textit{con}_n\} \\ \Gamma \vdash \textit{expr}' \leadsto \textit{exp}' : \textit{con}' \\ \Gamma \vdash \textit{con}_p : \textit{knd}_p \qquad \Gamma \vdash \textit{con}' \equiv \textit{con}_i[\textit{con}_p/\textit{cvar}_p] : \mathbf{T}\end{array}}{\Gamma \vdash \textit{longexpid}\ \textit{expr}' \leadsto \textit{pmod}.\overline{\textit{conid}}\_\text{in}[\textit{con}_p]\langle\!\langle \mathsf{inj}_{\textit{lab}_i}^{\textit{con}^{\text{sum}}[\textit{con}_p/\textit{cvar}_p]}\ \textit{exp}'\rangle\!\rangle : \textit{con}[\textit{con}_p/\textit{cvar}_p]} \tag{9.20}$$

Rule 9.20: Application of a (potentially polymorphic) `datatype` constructor. The rule is a bit complicated only because we make the optimization of inlining the constructor application as an injection into the appropriate sum type, followed by a call to the `datatype`'s in coercion.

$$\frac{\begin{array}{c}\Gamma \vdash \textit{expr} \leadsto \textit{exp} : \textit{con} \\ \Gamma \vdash \textit{ty} \leadsto \textit{con}' : \mathbf{T} \qquad \Gamma \vdash \textit{con} \equiv \textit{con}' : \mathbf{T}\end{array}}{\Gamma \vdash \textit{expr}\ \texttt{:}\ \textit{ty} \leadsto \textit{exp} : \textit{con}} \tag{9.21}$$

Rule 9.21: Type constraints on expressions are verified, but do not appear in the translation.

$$\frac{\Gamma \vdash \textit{expr} \leadsto \textit{exp} : \textit{con} \qquad \Gamma \vdash \textit{match} \leadsto \textit{val} : \mathsf{Tagged} \to \textit{con}' \qquad \Gamma \vdash \textit{con} \equiv \textit{con}' : \mathbf{T}}{\begin{array}{l}\Gamma \vdash \textit{expr}\ \texttt{handle}\ \textit{match} \leadsto \\ \quad \mathsf{handle}\ \textit{exp}\ \mathsf{with}\ \lambda(\textit{evar}{:}\mathsf{Tagged}){:}\textit{con}.(\mathsf{catch}^{\textit{con}}\ \textit{val}\ \textit{evar}\ \mathsf{with}\ \mathsf{raise}^{\textit{con}}\ \textit{evar}) : \textit{con}\end{array}} \tag{9.22}$$

Rule 9.22: The handling expression *val evar* will fail if the handler pattern does not match the exception caught by the IL handle, in which case we re-raise the exception.

$$\frac{\Gamma \vdash \textit{expr} \leadsto \textit{exp} : \mathsf{Tagged} \qquad \Gamma \vdash \textit{con} : \mathbf{T}}{\Gamma \vdash \texttt{raise}\ \textit{expr} \leadsto \mathsf{raise}^{\textit{con}}\ \textit{exp} : \textit{con}} \tag{9.23}$$

Rule 9.23: `raise` expressions can be given any (valid) type.

$$\frac{\Gamma \vdash \textit{match} \leadsto \textit{val} : \textit{con}_1 \to \textit{con}_2}{\begin{array}{l}\Gamma \vdash \texttt{fn}\ \textit{match} \leadsto \\ \quad \lambda(\textit{evar}{:}\textit{con}_1){:}\textit{con}_2.(\mathsf{catch}^{\textit{con}_2}\ \textit{val}\ \textit{evar}\ \mathsf{with}\ \mathsf{raise}^{\textit{con}_2}\ \textit{basis}.\overline{\texttt{Match}}^*.\overline{\texttt{Match}}) : \\ \quad \textit{con}_1 \to \textit{con}_2\end{array}} \tag{9.24}$$

Rule 9.24: The application *exp var* will fail if the match fails; we turn the failure into a match exception.

$$\frac{\begin{array}{c}\Gamma \vdash \textit{modexp} \leadsto \textit{mod} :_\kappa \textit{sig} \qquad \Gamma \vdash_{\text{ctx}} \textit{longconid} \leadsto \textit{vpmod}.\textit{lab} : \mathbf{T} \\ \Gamma \vdash \textit{vpmod}.\textit{lab}\_\text{pack} :_{\mathsf{P}} \textit{sig}' \xrightarrow{\mathsf{par}} [\![\text{it}{:}\textit{vpmod}.\textit{lab}]\!] \\ \Gamma, \textit{mvar}{:}\textit{sig} \vdash_{\text{sub}} \textit{mvar} : \textit{sig} \preceq \textit{sig}' \leadsto \textit{pmod}' : \_\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{pack}\ \textit{modexp}\ \texttt{as}\ \textit{longconid} \leadsto \\ \quad \mathsf{let}\ \textit{mvar}{=}\textit{mod}\ \mathsf{in}\ (\textit{vpmod}.\textit{lab}\_\text{pack}^{\mathsf{par}}(\textit{pmod}').\text{it} : \textit{vpmod}.\textit{lab}) : \textit{vpmod}.\textit{lab}\end{array}} \tag{9.25}$$

Rule 9.25: The package type is required to be a *longconid* so that the "pack" functor is easy to locate.

Matches

$$\boxed{\Gamma \vdash \mathit{match} \rightsquigarrow \mathit{val} : \mathit{con}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{con}' : \mathbf{T} \\ \Gamma, \mathit{evar}{:}\mathit{con}' \vdash \mathit{pat} \Leftarrow \mathit{evar} : \mathit{con}' \;\mathsf{else}\; \mathit{basis}.\mathrm{fail}^*.\mathrm{fail} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs} \\ \Gamma, \mathit{ilab}^*{\triangleright}\mathit{mvar}{:}[\![\mathit{ldecs}]\!] \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con}\end{array}}{\Gamma \vdash \mathit{pat}\; \texttt{=>}\; \mathit{expr} \rightsquigarrow \lambda(\mathit{evar}{:}\mathit{con}'){:}\mathit{con}.\mathsf{let}\; \mathit{mvar}{=}[\mathit{lbnds}]\; \mathsf{in}\; (\mathit{exp} : \mathit{con}) : \mathit{con}' \to \mathit{con}} \tag{9.26}$$

Rule 9.26: The type *con* is well-formed in Γ because the *ldecs* only contain term declarations.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{mrule} \rightsquigarrow \mathit{val} : \mathit{con}_1 \to \mathit{con}_2 \\ \Gamma \vdash \mathit{match} \rightsquigarrow \mathit{val}' : \mathit{con}_1' \to \mathit{con}_2' \\ \Gamma \vdash \mathit{con}_1 \to \mathit{con}_2 \equiv \mathit{con}_1' \to \mathit{con}_2' : \mathbf{T}\end{array}}{\Gamma \vdash \mathit{mrule}\; \texttt{|}\; \mathit{match} \rightsquigarrow \lambda(\mathit{evar}{:}\mathit{con}_1){:}\mathit{con}_2.\mathsf{catch}^{\mathit{con}_2}\; \mathit{val}\; \mathit{evar}\; \mathsf{with}\; \mathit{val}'\; \mathit{evar} : \mathit{con}_1 \to \mathit{con}_2} \tag{9.27}$$

Bindings

$$\boxed{\Gamma \vdash \mathit{bindings} \rightsquigarrow \mathit{lbnds} :_\kappa \mathit{ldecs}}$$

$$\frac{}{\Gamma \vdash \cdot \rightsquigarrow \cdot :_{\mathrm{P}} \cdot} \tag{9.28}$$

$$\frac{\Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig} \qquad \Gamma, \overline{\mathit{sigid}}{=}\mathit{sig} \vdash \mathit{bindings} \rightsquigarrow \mathit{lbnds} :_\kappa \mathit{ldecs}}{\Gamma \vdash \texttt{signature}\; \mathit{sigid}\; \texttt{=}\; \mathit{sigexp}\; \langle\texttt{;}\rangle\; \mathit{bindings} \rightsquigarrow \mathit{lbnds} :_\kappa \mathit{ldecs}} \tag{9.29}$$

Rule 9.29: `signature` bindings do not produce any actual IL bindings or declarations; they just provide shorthand for signature expressions during elaboration.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{binding} \rightsquigarrow \mathit{lbnds}_1 :_{\kappa_1} \mathit{ldecs}_1 \\ \Gamma, \mathit{ldecs}_1 \vdash \mathit{bindings} \rightsquigarrow \mathit{lbnds}_2 :_{\kappa_2} \mathit{ldecs}_2\end{array}}{\Gamma \vdash \mathit{binding}\; \langle\texttt{;}\rangle\; \mathit{bindings} \rightsquigarrow \mathit{lbnds}_1 {+\!\!+} \mathit{lbnds}_2 :_{\kappa_1 \sqcup \kappa_2} \mathit{ldecs}_1 {+\!\!+} \mathit{ldecs}_2} \tag{9.30}$$

Rule 9.30: The syntactic-concatenation-with-renaming operation is used here to relabel any shadowed bindings from lbnds_1.

$$\boxed{\Gamma \vdash \mathit{binding} \rightsquigarrow \mathit{lbnds} :_\kappa \mathit{ldecs}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con} \\ \Gamma, \mathit{evar}{:}\mathit{con} \vdash \mathit{pat} \Leftarrow \mathit{evar} : \mathit{con}\; \mathsf{else}\; \mathit{basis}.\overline{\texttt{Bind}}^*.\overline{\texttt{Bind}} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}\end{array}}{\Gamma \vdash \texttt{val}\; \mathit{pat}\; \texttt{=}\; \mathit{expr} \rightsquigarrow \mathit{ilab}{\triangleright}\mathit{evar}{=}\mathit{exp}, \mathit{lbnds} :_{\mathrm{P}} \mathit{ilab}{\triangleright}\mathit{evar}{:}\mathit{con}, \mathit{ldecs}} \tag{9.31}$$

Rule 9.31: Monomorphic `val` bindings. If the pattern match fails, we catch the fail exception and reraise a bind exception. We separate monomorphic `val` bindings from polymorphic `val` bindings because polymorphic bindings impose a value restriction on *exp* while monomorphic bindings do not.

$$\frac{\begin{array}{l} \mathit{knd}_p := [\![\overline{\mathit{tyvar}_1}{:}\mathbf{T}, \cdots, \overline{\mathit{tyvar}_m}{:}\mathbf{T}, 1{:}\mathbf{T}, \cdots, k{:}\mathbf{T}]\!] \\ \Gamma, \mathit{ilab}^*{\triangleright}\mathit{mvar}_p{:}\mathit{knd}_p \vdash \mathit{expr} \rightsquigarrow \mathit{vexp} : \mathit{con} \\ \Gamma' := \Gamma, \mathit{ilab}^*{\triangleright}\mathit{mvar}_p{:}\mathit{knd}_p, \mathit{evar}{:}\forall(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{con} \\ \Gamma' \vdash \mathit{pat} \Leftarrow \mathit{evar}[\mathit{mvar}_p^{\mathsf{c}}] : \mathit{con}\ \mathsf{else}\ \mathit{basis}.\overline{\mathtt{Bind}}^*.\overline{\mathtt{Bind}} \rightsquigarrow \\ \qquad \mathit{lab}_1{=}\mathit{vexp}_1, \cdots, \mathit{lab}_n{=}\mathit{vexp}_n : \mathit{lab}_1{:}\mathit{con}_1, \cdots, \mathit{lab}_n{:}\mathit{con}_n \\ \forall i \in 1..n : \\ \qquad \mathit{lbnd}_i := \mathit{lab}_i{=}\Lambda(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{vexp}_i \\ \qquad \mathit{ldec}_i := \mathit{lab}_i{:}\forall(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{con}_i \end{array}}{\begin{array}{l} \Gamma \vdash \mathtt{val}\ \mathtt{(}\mathit{tyvar}_1\mathtt{,}\cdots\mathtt{,}\mathit{tyvar}_m\mathtt{)}\ \mathit{pat}\ \mathtt{=}\ \mathit{expr} \rightsquigarrow \\ \qquad \mathit{ilab}{\triangleright}\mathit{evar}{=}\Lambda(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{vexp}, \mathit{lbnd}_1, \cdots, \mathit{lbnd}_n :_{\mathtt{P}} \\ \qquad \mathit{ilab}{\triangleright}\mathit{evar}{:}\forall(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{con}, \mathit{ldec}_1, \cdots, \mathit{ldec}_n \end{array}} \quad (9.32)$$

Rule 9.32: Polymorphic `val` bindings. Like HS, I assume that a `val` declaration is explicitly annotated with the type variables scoped in that declaration. Type inference may introduce k additional type variables that are not mentioned in the source (as in `val f = fn x => x`). The translation of *expr* is required to be valuable, so that the type abstraction in the output of the rule is well-formed and does not suspend any computational effects. I require all the pattern-matching bindings to be valuable as well; this means effectively that *pat* must be irrefutable and not contain any `ref` patterns. (This requirement marks a departure from HS, in which some of the pattern bindings are allowed to be non-valuable and are consequently *not* generalized. I believe my semantics is easier to understand, and it conforms more closely to what is implemented in TILT.)

$$\frac{\begin{array}{l} \mathit{knd}_p := [\![\overline{\mathit{tyvar}_1}{:}\mathbf{T}, \cdots, \overline{\mathit{tyvar}_m}{:}\mathbf{T}, 1{:}\mathbf{T}, \cdots, k{:}\mathbf{T}]\!] \\ \Gamma' := \Gamma, \mathit{ilab}^*{\triangleright}\mathit{mvar}_p{:}\mathit{knd}_p, \overline{\mathit{expid}_1}{\triangleright}\mathit{evar}'_1{:}\mathit{con}_1 \to \mathit{con}'_1, \cdots, \overline{\mathit{expid}_n}{\triangleright}\mathit{evar}'_n{:}\mathit{con}_n \to \mathit{con}'_n \\ \forall i \in 1..n : \quad \Gamma' \vdash \mathit{match}_i \rightsquigarrow \lambda(\mathit{evar}_i{:}\mathit{con}_i){:}\mathit{con}'_i.\mathit{exp}_i : \mathit{con}_i \to \mathit{con}'_i \\ \mathit{val}_k := \mathsf{fix}_k\ (\mathit{evar}'_i(\mathit{evar}_i{:}\mathit{con}_i){:}\mathit{con}'_i{=}\mathit{exp}_i)_{i=1}^n\ \mathsf{end} \end{array}}{\begin{array}{l} \Gamma \vdash \mathtt{fun}\ \mathtt{(}\mathit{tyvar}_1\mathtt{,}\cdots\mathtt{,}\mathit{tyvar}_m\mathtt{)}\ \mathit{expid}_1\ \mathtt{=}\ \mathtt{fn}\ \mathit{match}_1\ \mathtt{and}\ \cdots\ \mathtt{and}\ \mathit{expid}_n\ \mathtt{=}\ \mathtt{fn}\ \mathit{match}_n \rightsquigarrow \\ \qquad \overline{\mathit{expid}_1}{=}\Lambda(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{val}_1, \cdots, \overline{\mathit{expid}_n}{=}\Lambda(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{val}_n :_{\mathtt{P}} \\ \qquad \overline{\mathit{expid}_1}{:}\forall(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{con}_1 \to \mathit{con}'_1, \cdots, \overline{\mathit{expid}_n}{:}\forall(\mathit{mvar}_p^{\mathsf{c}}{:}\mathit{knd}_p).\mathit{con}_n \to \mathit{con}'_n \end{array}} \quad (9.33)$$

Rule 9.33: My syntax for `fun` bindings is slightly different from SML's. It would not be difficult to support SML-style clausal function definitions by desugaring them into the present syntax.

$$\frac{\forall i \in 1..n : \quad \Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longmodid}_i} \rightsquigarrow \mathit{pmod}_i : \mathit{tsig}_i}{\begin{array}{l} \Gamma \vdash \mathtt{open}\ \mathit{longmodid}_1\ \cdots\ \mathit{longmodid}_n \rightsquigarrow \\ \qquad \mathit{ilab}_1{}^*{=}\mathit{pmod}_1, \cdots, \mathit{ilab}_n{}^*{=}\mathit{pmod}_n :_{\mathtt{P}} \mathit{ilab}_1{}^*{:}\mathit{tsig}_1, \cdots, \mathit{ilab}_n{}^*{:}\mathit{tsig}_n \end{array}} \quad (9.34)$$

Rule 9.34: Makes the components of the opened modules visible in the current namespace by binding with open labels. The signatures of the modules are specified as *tsig*'s simply to emphasize that the signatures of projectible modules are transparent.

$$\frac{\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \mathbf{T}}{\begin{array}{l} \Gamma \vdash \mathtt{exception}\ \mathit{expid}\ \mathtt{of}\ \mathit{ty} \rightsquigarrow \\ \qquad \mathit{ilab}^*{=}[\mathrm{tag}{\triangleright}\mathit{evar}{=}\mathsf{new\_tag}[\mathit{con}], \overline{\mathit{expid}}{=}\lambda(\mathit{evar}'{:}\mathit{con}){:}\mathsf{Tagged}.\mathsf{tag}(\mathit{evar}, \mathit{evar}')] :_{\mathtt{P}} \\ \qquad \mathit{ilab}^*{:}[\![\mathrm{tag}{\triangleright}\mathit{evar}{:}\mathit{con}\ \mathsf{Tag}, \overline{\mathit{expid}}{:}\mathit{con} \to \mathsf{Tagged}]\!] \end{array}} \quad (9.35)$$

$$\frac{}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ \mathit{expid} \leadsto \\ \quad \mathit{ilab}^*{=}[\mathrm{tag}{\triangleright}\mathit{evar}{=}\mathsf{new\_tag}[\mathsf{Unit}], \overline{\mathit{expid}}{=}\mathsf{tag}(\mathit{evar}, \{\})] :_{\mathrm{P}} \\ \quad \mathit{ilab}^*{:}[\![\mathrm{tag}{\triangleright}\mathit{evar}{:}\mathsf{Unit\ Tag}, \overline{\mathit{expid}}{:}\mathsf{Tagged}]\!]\end{array}} \tag{9.36}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longexpid}} \leadsto \mathit{pmod}.\mathit{lab} : \mathit{con} \\ \Gamma \vdash \mathit{pmod}.\mathrm{tag} : \mathit{con}'\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{exception}\ \mathit{expid}\ \texttt{=}\ \mathit{longexpid} \leadsto \\ \quad \mathit{ilab}^*{=}[\mathrm{tag}{=}\mathit{pmod}.\mathrm{tag}, \overline{\mathit{expid}}{=}\mathit{pmod}.\mathit{lab}] :_{\mathrm{P}} \mathit{ilab}^*{:}[\![\mathrm{tag}{:}\mathit{con}', \overline{\mathit{expid}}{:}\mathit{con}]\!]\end{array}} \tag{9.37}$$

Rule 9.37: Structures containing a "tag" component are created by EL `exception` declarations only.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{bindings}_1 \leadsto \mathit{lbnds}_1 :_{\kappa_1} \mathit{ldecs}_1 \\ \Gamma, \mathit{ilab}^*{\triangleright}\mathit{mvar}{:}[\![\mathit{ldecs}_1]\!] \vdash \mathit{bindings}_2 \leadsto \mathit{lbnds}_2 :_{\kappa_2} \mathit{ldecs}_2\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{local}\ \mathit{bindings}_1\ \texttt{in}\ \mathit{bindings}_2\ \texttt{end} \leadsto \\ \quad \mathit{ilab}{\triangleright}\mathit{mvar}{=}[\mathit{lbnds}_1], \mathit{lbnds}_2 :_{\kappa_1 \sqcup \kappa_2} \mathit{ilab}{\triangleright}\mathit{mvar}{:}[\![\mathit{ldecs}_1]\!], \mathit{ldecs}_2\end{array}} \tag{9.38}$$

Rule 9.38: This rule exemplifies HS's approach to the avoidance problem, which I have followed. Bindings are exported for all of the declarations, so it is perfectly OK for the public ldecs_2 to refer to the private ldecs_1 (since the ldecs_1 do not go out of scope). The ldecs_1 are guaranteed to be inaccessible from the EL, however, because they are segregated into a substructure with an internal label (that is not open).

$$\frac{\Gamma, \overline{\mathit{tyvar}_1}{\triangleright}\mathit{cvar}_1{:}\mathbf{T}, \cdots, \overline{\mathit{tyvar}_n}{\triangleright}\mathit{cvar}_n{:}\mathbf{T} \vdash \mathit{ty} \leadsto \mathit{con} : \mathbf{T}}{\begin{array}{l}\Gamma \vdash \texttt{type (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{conid}\ \texttt{=}\ \mathit{ty} \leadsto \\ \quad \overline{\mathit{conid}}{=}\lambda(\mathit{cvar}_1, \cdots, \mathit{cvar}_n).\mathit{con} :_{\mathrm{P}} \overline{\mathit{conid}}{:}\Pi(\mathit{cvar}_1, \cdots, \mathit{cvar}_n).\mathfrak{S}(\mathit{con})\end{array}} \tag{9.39}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longconid}} \leadsto \mathit{pmod}.\mathit{lab} : \mathit{tknd} \\ \Gamma \vdash \mathit{pmod} :_{\mathrm{P}} [\![\mathit{lab}{\triangleright}\mathit{cvar}{:}\mathit{tknd}, \mathit{lab}\_\mathrm{in}{:}\mathit{con}_{\mathrm{in}}, \mathit{lab}\_\mathrm{out}{:}\mathit{con}_{\mathrm{out}}, \mathit{ldecs}]\!] \\ \mathit{ldecs} = \mathit{lab}_1{\triangleright}\mathit{dec}_1, \cdots, \mathit{lab}_n{\triangleright}\mathit{dec}_n\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{datatype}\ \mathit{conid}\ \texttt{=}\ \texttt{datatype}\ \mathit{longconid} \leadsto \\ \quad \mathit{ilab}^*{=}\left[\begin{array}{l}\overline{\mathit{conid}}{\triangleright}\mathit{cvar}{=}\mathit{pmod}.\mathit{lab}, \overline{\mathit{conid}}\_\mathrm{in}{=}\mathit{pmod}.\mathit{lab}\_\mathrm{in}, \\ \overline{\mathit{conid}}\_\mathrm{out}{=}\mathit{pmod}.\mathit{lab}\_\mathrm{out}, \mathit{lab}_1{=}\mathit{pmod}.\mathit{lab}_1, \cdots, \mathit{lab}_n{=}\mathit{pmod}.\mathit{lab}_n\end{array}\right] :_{\mathrm{P}} \\ \quad \mathit{ilab}^*{:}[\![\overline{\mathit{conid}}{\triangleright}\mathit{cvar}{:}\mathit{tknd}, \overline{\mathit{conid}}\_\mathrm{in}{:}\mathit{con}_{\mathrm{in}}, \overline{\mathit{conid}}\_\mathrm{out}{:}\mathit{con}_{\mathrm{out}}, \mathit{ldecs}]\!]\end{array}} \tag{9.40}$$

Rule 9.40: Copying a `datatype` essentially involves copying the whole module that defines the `datatype`, its in and out coercions, and its data constructors. Unfortunately, we cannot just copy the whole module all at once because the name of the new type affects the labels assigned to the type component and the in and out coercions.

$$\frac{\Gamma \vdash \mathit{datbinds} \leadsto \mathit{sig} \qquad \Gamma \vdash_{\mathrm{can}} \mathit{sig} \leadsto \mathit{mod}}{\Gamma \vdash \texttt{datatype}\ \mathit{datbinds} \leadsto \mathit{ilab}^*{=}(\mathit{mod} :>_{\mathrm{P}} \mathit{sig}) : \mathit{ilab}^*{:}\mathit{sig}} \tag{9.41}$$

Rule 9.41: For `datatype` bindings, we compute the canonical implementation of the corresponding `datatype` spec.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \mathit{mod} := \left[\begin{array}{l}\overline{\mathit{conid}}\triangleright \mathit{cvar}{=}\langle\!|\mathit{sig}|\!\rangle, \\ \overline{\mathit{conid}}\_\mathrm{pack}{=}\lambda^{\mathsf{par}}(\mathit{mvar}{:}\mathit{sig}){:}[\![\mathrm{it}{:}\mathit{cvar}]\!].[\mathrm{it}{=}\,\mathsf{pack}\ \mathit{mvar}\ \mathsf{as}\ \mathit{sig}], \\ \overline{\mathit{conid}}\_\mathrm{unpack}{=}\lambda^{\mathsf{par}}(\mathit{mvar}{:}[\![\mathrm{it}{:}\mathit{cvar}]\!]){:}\mathit{sig}.\mathsf{unpack}\ \mathit{mvar}.\mathrm{it}\ \mathsf{as}\ \mathit{sig}\end{array}\right] \\ \mathit{sig}' := \left[\!\!\left[\begin{array}{l}\overline{\mathit{conid}}\triangleright \mathit{cvar}{:}\mathbf{T}, \\ \overline{\mathit{conid}}\_\mathrm{pack}{:}\mathit{sig} \xrightarrow{\mathsf{par}} [\![\mathrm{it}{:}\mathit{cvar}]\!], \\ \overline{\mathit{conid}}\_\mathrm{unpack}{:}[\![\mathrm{it}{:}\mathit{cvar}]\!] \xrightarrow{\mathsf{par}} \mathit{sig}\end{array}\right]\!\!\right]\end{array}}{\Gamma \vdash \texttt{packtype}\ \mathit{conid}\ \texttt{=}\ \mathit{sigexp} \leadsto \mathit{ilab}^*{=}(\mathit{mod} :>_{\mathsf{P}} \mathit{sig}') :_{\mathsf{P}} \mathit{ilab}^*{:}\mathit{sig}'} \quad (9.42)$$

Rule 9.42: It is necessary that the "unpack" functor be partial, since unpacking is an impure/inseparable operation. It does not matter whether the "pack" functor is total or partial, since it is only ever applied by Rule 9.25.

$$\frac{\Gamma \vdash \mathit{modexp} \leadsto \mathit{mod} :_{\kappa} \mathit{sig}}{\Gamma \vdash \texttt{module}\ \mathit{modid}\ \texttt{=}\ \mathit{modexp} \leadsto \overline{\mathit{modid}}{=}\mathit{mod} :_{\kappa} \overline{\mathit{modid}}{:}\mathit{sig}} \quad (9.43)$$

Module Expressions

$\boxed{\Gamma \vdash \mathit{modexp} \leadsto \mathit{mod} :_{\kappa} \mathit{sig}}$

Note: The translation rule for recursive modules is presented in Section 9.3.8.

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longmodid}} \leadsto \mathit{pmod} : \mathit{tsig}}{\Gamma \vdash \mathit{longmodid} \leadsto \mathit{pmod} :_{\mathsf{P}} \mathit{tsig}} \quad (9.44)$$

$$\frac{\Gamma \vdash \mathit{bindings} \leadsto \mathit{lbnds} :_{\kappa} \mathit{ldecs}}{\Gamma \vdash \texttt{struct}\ \mathit{bindings}\ \texttt{end} \leadsto [\mathit{lbnds}] :_{\kappa} [\![\mathit{ldecs}]\!]} \quad (9.45)$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{modexp} \leadsto \mathit{mod} :_{\kappa} \mathit{sig} \\ \Gamma, \mathit{mvar}{:}\mathit{sig}; \mathit{mvar}{:}\mathit{sig} \vdash_{\overline{\mathrm{sig}}} \overline{\mathit{modid}} \leadsto \mathit{pmod} : \mathit{tsig} \\ \mathit{modexp}\ \text{is not of the form}\ \mathit{longmodid}\end{array}}{\Gamma \vdash \mathit{modexp}\,\texttt{.}\,\mathit{modid} \leadsto \mathsf{elet}\ \mathit{mvar}{=}\mathit{mod}\ \mathsf{in}\ \mathit{pmod} :_{\kappa} \exists(\mathit{mvar}{:}\mathit{sig}).\mathit{tsig}} \quad (9.46)$$

Rule 9.46: $\mathit{modexp}.\mathit{modid}$ is elaborated as if it were `let module` $\mathit{modid}' = \mathit{modexp}$ `in` $\mathit{modid}'.\mathit{modid}$ `end`. See the elaboration of module-level `let` bindings (Rule 9.50 below).

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \Gamma, \overline{\mathit{modid}}\triangleright \mathit{mvar}{:}\mathit{sig} \vdash \mathit{modexp} \leadsto \mathit{mod} :_{\mathsf{P}} \mathit{sig}'\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{functor (}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{) ->}\ \mathit{modexp} \leadsto \\ \qquad \lambda^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{mod} :_{\mathsf{P}} \Pi^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{sig}'\end{array}} \quad (9.47)$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \Gamma, \overline{\mathit{modid}}\triangleright \mathit{mvar}{:}\mathit{sig} \vdash \mathit{modexp} \leadsto \mathit{mod} :_{\kappa} \mathit{sig}'\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{functor (}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{) ->>}\ \mathit{modexp} \leadsto \\ \qquad \lambda^{\mathsf{par}}(\mathit{mvar}{:}\mathit{sig}){:}\mathit{sig}'.\mathit{mod} :_{\mathsf{P}} \Pi^{\mathsf{par}}(\mathit{mvar}{:}\mathit{sig}).\mathit{sig}'\end{array}} \quad (9.48)$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{modexp}_1 \rightsquigarrow \mathit{mod}_1 :_{\kappa_1} \mathit{sig}_1 \\ \Gamma \vdash \mathit{modexp}_2 \rightsquigarrow \mathit{mod}_2 :_{\kappa_2} \mathit{sig}_2 \\ \Gamma' := \Gamma, \mathit{mvar}_1{:}\mathit{sig}_1, \mathit{mvar}_2{:}\mathit{sig}_2 \\ \Gamma' \vDash_{\text{peel}} \mathit{mvar}_1{:}\mathit{sig}_1 \rightsquigarrow \mathit{pmod} : \Pi^\tau(\mathit{mvar}{:}\mathit{sig}').\mathit{sig}'' \\ \Gamma' \vDash_{\text{sub}} \mathit{mvar}_2 : \mathit{sig}_2 \preceq \mathit{sig}' \rightsquigarrow \mathit{pmod}' : \_ \\ \kappa = \begin{cases} \mathtt{P} & \text{if } \kappa_1 = \kappa_2 = \mathtt{P} \text{ and } \tau = \mathsf{tot} \\ \mathtt{I} & \text{otherwise} \end{cases}\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{modexp}_1\mathtt{(}\mathit{modexp}_2\mathtt{)} \rightsquigarrow \\ \quad \mathsf{elet}\ \mathit{mvar}_1{=}\mathit{mod}_1\ \mathsf{in}\ \mathsf{elet}\ \mathit{mvar}_2{=}\mathit{mod}_2\ \mathsf{in}\ \mathit{pmod}^\tau(\mathit{pmod}') :_\kappa \\ \quad \exists(\mathit{mvar}_1{:}\mathit{sig}_1).\exists(\mathit{mvar}_2{:}\mathit{sig}_2).\mathit{sig}''[\mathit{pmod}'/\mathit{mvar}]\end{array}} \tag{9.49}$$

Rule 9.49: Note the use of peeling to uncover the underlying functor signature of mod_1.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{bindings} \rightsquigarrow \mathit{lbnds} :_{\kappa_1} \mathit{ldecs} \\ \Gamma, \mathit{ilab}^*{\triangleright}\mathit{mvar}{:}[\![\mathit{ldecs}]\!] \vdash \mathit{modexp} \rightsquigarrow \mathit{mod} :_{\kappa_2} \mathit{sig}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{let}\ \mathit{bindings}\ \mathtt{in}\ \mathit{modexp}\ \mathtt{end} \rightsquigarrow \\ \quad \mathsf{elet}\ \mathit{mvar}{=}[\mathit{lbnds}]\ \mathsf{in}\ \mathit{mod} :_{\kappa_1 \sqcup \kappa_2} \exists(\mathit{mvar}{:}[\![\mathit{ldecs}]\!]).\mathit{sig}\end{array}} \tag{9.50}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{modexp} \rightsquigarrow \mathit{mod} :_\kappa \mathit{sig} \qquad \Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{tsig} : \mathsf{Sig} \\ \Gamma, \mathit{mvar}{:}\mathit{sig} \vDash_{\text{sub}} \mathit{mvar} : \mathit{sig} \preceq \mathit{tsig} \rightsquigarrow \mathit{pmod} : \_\end{array}}{\Gamma \vdash \mathit{modexp}\ \ \mathit{seal}\ \ \mathit{sigexp} \rightsquigarrow \mathsf{purify}(\mathsf{let}\ \mathit{mvar}{=}\mathit{mod}\ \mathsf{in}\ (\mathit{pmod} : \mathit{tsig})) :_{\mathtt{P}} \mathit{tsig}} \tag{9.51}$$

Rule 9.51: Regardless of what kind of sealing is requested, if the sealing signature is transparent, then we take this chance to observe that the sealed module is in fact pure/separable.

$$\frac{\begin{array}{c}\text{Rule 9.51 does not apply.} \\ \Gamma \vdash \mathit{modexp} \rightsquigarrow \mathit{mod} :_\kappa \mathit{sig} \qquad \Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig}' : \mathsf{Sig} \\ \Gamma, \mathit{mvar}{:}\mathit{sig} \vDash_{\text{sub}} \mathit{mvar} : \mathit{sig} \preceq \mathit{sig}' \rightsquigarrow \mathit{pmod} : \mathit{tsig}\end{array}}{\Gamma \vdash \mathit{modexp}\ \ \mathtt{:}\ \ \mathit{sigexp} \rightsquigarrow \mathsf{elet}\ \mathit{mvar}{=}\mathit{mod}\ \mathsf{in}\ \mathit{pmod} :_\kappa \exists(\mathit{mvar}{:}\mathit{sig}).\mathit{tsig}} \tag{9.52}$$

Rule 9.52: As in SML, ascribing a signature to a structure using "`:`" hides components (this hiding being accomplished via an explicit coercion), but allows the identity of the remaining type components to leak through in the transparent *tsig*.

$$\frac{\begin{array}{c}\text{Rule 9.51 does not apply.} \\ \Gamma \vdash \mathit{modexp} \rightsquigarrow \mathit{mod} :_\kappa \mathit{sig} \qquad \Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig}' : \mathsf{Sig} \\ \Gamma, \mathit{mvar}{:}\mathit{sig} \vDash_{\text{sub}} \mathit{mvar} : \mathit{sig} \preceq \mathit{sig}' \rightsquigarrow \mathit{pmod} : \_\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{modexp}\ \ \mathtt{:>}\ \mathit{sigexp} \rightsquigarrow \\ \quad \mathsf{let}\ \mathit{mvar}{=}\mathit{mod}\ \mathsf{in}\ ((\mathit{pmod} \mathrel{\mathtt{:>}_{\mathtt{P}}} \mathit{sig}') : \mathit{sig}') :_\kappa \mathit{sig}'\end{array}} \tag{9.53}$$

$$\frac{\begin{array}{c}\text{Rule 9.51 does not apply.} \\ \Gamma \vdash \mathit{modexp} \rightsquigarrow \mathit{mod} :_\kappa \mathit{sig} \qquad \Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig}' : \mathsf{Sig} \\ \Gamma, \mathit{mvar}{:}\mathit{sig} \vDash_{\text{sub}} \mathit{mvar} : \mathit{sig} \preceq \mathit{sig}' \rightsquigarrow \mathit{pmod} : \_\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{modexp}\ \ \mathtt{:>>}\ \mathit{sigexp} \rightsquigarrow \\ \quad \mathsf{let}\ \mathit{mvar}{=}\mathit{mod}\ \mathsf{in}\ ((\mathit{pmod} \mathrel{\mathtt{:>}_{\mathtt{I}}} \mathit{sig}') : \mathit{sig}') :_{\mathtt{I}} \mathit{sig}'\end{array}} \tag{9.54}$$

Rules 9.53 and 9.54: Opaque sealing does not require the use of an existential signature; we can just use a normal IL module-level let, as the result is sealed with a signature that is well-formed in Γ. The only difference between basic and impure sealing is which IL sealing construct they translate into.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{expr} \rightsquigarrow \mathit{exp} : \mathit{con} \qquad \Gamma \vdash_{\mathrm{ctx}} \mathit{longconid} \rightsquigarrow \mathit{vpmod}.\mathit{lab} : \mathbf{T} \\ \Gamma \vdash \mathit{con} \equiv \mathit{vpmod}.\mathit{lab} : \mathbf{T} \qquad \Gamma \vdash \mathit{vpmod}.\mathit{lab}\_\mathrm{unpack} :_{\mathtt{P}} [\![\mathrm{it}{:}\mathit{con}]\!] \xrightarrow{\mathsf{par}} \mathit{sig}\end{array}}{\Gamma \vdash \mathtt{unpack}\ \mathit{expr}\ \mathtt{as}\ \mathit{longconid} \rightsquigarrow \mathit{vpmod}.\mathit{lab}\_\mathrm{unpack}^{\mathsf{par}}([\mathrm{it}{=}\mathit{exp}]) :_{\mathtt{I}} \mathit{sig}} \tag{9.55}$$

Specifications

$$\boxed{\begin{array}{r}\Gamma \vdash \mathit{specs} \rightsquigarrow \mathit{ldecs} \\ \Gamma \vdash_{\overline{\mathrm{stat}}} \mathit{specs} \rightsquigarrow \mathit{statldecs}\end{array}}$$

Normal and static elaboration rules for specifications and signature expressions are often very similar. I will write $\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{spec}(s) \rightsquigarrow \mathit{ldecs}$ and $\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig}$ in certain rules to denote that the rule can be used to derive either judgment by replacing all instances of $\vdash_{\langle\mathrm{stat}\rangle}$ with $\vdash$ or all instances of $\vdash_{\langle\mathrm{stat}\rangle}$ with $\vdash_{\overline{\mathrm{stat}}}$.

$$\frac{}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \cdot \rightsquigarrow \cdot} \tag{9.56}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{spec} \rightsquigarrow \mathit{ldecs}_1 \qquad \Gamma, \mathit{ldecs}_1 \vdash_{\langle\mathrm{stat}\rangle} \mathit{specs} \rightsquigarrow \mathit{ldecs}_2 \\ \mathrm{vislabs}(\mathit{ldecs}_1) \cap \mathrm{vislabs}(\mathit{ldecs}_2) = \emptyset\end{array}}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{spec}\ \mathit{specs} \rightsquigarrow \mathit{ldecs}_1, \mathit{ldecs}_2} \tag{9.57}$$

Rule 9.57: All that is necessary for the output *ldecs* to be well-formed is that labdom(ldecs_1) and labdom(ldecs_2) be disjoint. This condition would not, however, prevent one from writing a spec containing, say, two `datatype` specifications of the same type, due to the use of open labels in specs. We therefore require additionally that the *visible* labels exported by each spec be disjoint from the visible labels exported by all the other specs.

$$\boxed{\begin{array}{r}\Gamma \vdash \mathit{spec} \rightsquigarrow \mathit{ldecs} \\ \Gamma \vdash_{\overline{\mathrm{stat}}} \mathit{spec} \rightsquigarrow \mathit{statldecs}\end{array}}$$

$$\frac{\Gamma, \overline{\mathit{tyvar}_1} \triangleright \mathit{cvar}_1{:}\mathbf{T}, \cdots, \overline{\mathit{tyvar}_n} \triangleright \mathit{cvar}_n{:}\mathbf{T} \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \mathbf{T}}{\Gamma \vdash \mathtt{val}\ (\mathit{tyvar}_1, \cdots, \mathit{tyvar}_n)\ \mathit{expid}\ :\ \mathit{ty} \rightsquigarrow \overline{\mathit{expid}}{:}\forall(\mathit{cvar}_1, \cdots, \mathit{cvar}_n).\mathit{con}} \tag{9.58}$$

Rule 9.58: Specification of a (potentially polymorphic) value. As with polymorphic value *bindings*, we assume that the free type variables tyvar_i in *ty* are bound explicitly.

$$\frac{\mathit{spec} \text{ is a } \mathtt{val} \text{ specification.}}{\Gamma \vdash_{\overline{\mathrm{stat}}} \mathit{spec} \rightsquigarrow \cdot} \tag{9.59}$$

Rule 9.59: Static elaboration ignores `val` specs.

$$\frac{\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \mathbf{T}}{\Gamma \vdash \mathtt{exception}\ \mathit{expid}\ \mathtt{of}\ \mathit{ty} \rightsquigarrow \overline{\mathit{expid}}{:}[\![\mathrm{tag}{:}\mathit{con}\ \mathsf{Tag}, \overline{\mathit{expid}}{:}\mathit{con} \rightarrow \mathsf{Tagged}]\!]} \tag{9.60}$$

$$\frac{}{\Gamma \vdash \texttt{exception}\ \mathit{expid} \leadsto \overline{\mathit{expid}}{:}[\![\mathrm{tag}{:}\mathsf{Unit\,Tag}, \overline{\mathit{expid}}{:}\mathsf{Tagged}]\!]} \tag{9.61}$$

$$\frac{\mathit{spec}\text{ is of the form }\texttt{exception}\ \mathit{expid}\text{ or }\texttt{exception}\ \mathit{expid}\ \texttt{of}\ \mathit{ty}.}{\Gamma \vdash_{\mathrm{stat}} \mathit{spec} \leadsto \overline{\mathit{expid}}{:}[\![\cdot]\!]} \tag{9.62}$$

$$\frac{}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \texttt{type (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{expid} \leadsto \overline{\mathit{expid}}{:}\mathbf{T}^n{\to}\mathbf{T}} \tag{9.63}$$

$$\frac{\Gamma, \overline{\mathit{tyvar}_1}{\triangleright}\mathit{cvar}_1{:}\mathbf{T}, \cdots, \overline{\mathit{tyvar}_n}{\triangleright}\mathit{cvar}_n{:}\mathbf{T} \vdash \mathit{ty} \leadsto \mathit{con} : \mathbf{T}}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \texttt{type (}\mathit{tyvar}_1\texttt{,}\cdots\texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{expid}\ \texttt{=}\ \mathit{ty} \leadsto \overline{\mathit{expid}}{:}\Pi(\mathit{cvar}_1, \cdots, \mathit{cvar}_n).\mathfrak{S}(\mathit{con})} \tag{9.64}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longconid}} \leadsto \mathit{pmod}.\mathit{lab} : \mathit{tknd} \\ \Gamma \vdash \mathit{pmod} :_{\mathsf{P}} [\![\mathit{lab}{\triangleright}\mathit{cvar}{:}\mathit{tknd}, \mathit{lab}\_\mathrm{in}{:}\mathit{con}_{\mathrm{in}}, \mathit{lab}\_\mathrm{out}{:}\mathit{con}_{\mathrm{out}}, \mathit{ldecs}]\!]\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{datatype}\ \mathit{conid}\ \texttt{=}\ \texttt{datatype}\ \mathit{longconid} \leadsto \\ \qquad \mathit{ilab}^*{:}[\![\overline{\mathit{conid}}{\triangleright}\mathit{cvar}{:}\mathit{tknd}, \overline{\mathit{conid}}\_\mathrm{in}{:}\mathit{con}_{\mathrm{in}}, \overline{\mathit{conid}}\_\mathrm{out}{:}\mathit{con}_{\mathrm{out}}, \mathit{ldecs}]\!]\end{array}} \tag{9.65}$$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} \overline{\mathit{longconid}} \leadsto \mathit{con} : \mathit{tknd}}{\Gamma \vdash_{\mathrm{stat}} \texttt{datatype}\ \mathit{conid}\ \texttt{=}\ \texttt{datatype}\ \mathit{longconid} \leadsto \mathit{ilab}^*{:}[\![\overline{\mathit{conid}}{:}\mathit{tknd}]\!]} \tag{9.66}$$

$$\frac{\Gamma \vdash \mathit{datbinds} \leadsto \mathit{sig}}{\Gamma \vdash \texttt{datatype}\ \mathit{datbinds} \leadsto \mathit{ilab}^*{:}\mathit{sig}} \tag{9.67}$$

$$\frac{\Gamma \vdash \mathit{datbinds} \leadsto \mathit{sig}}{\Gamma \vdash_{\mathrm{stat}} \texttt{datatype}\ \mathit{datbinds} \leadsto \mathit{ilab}^*{:}\mathsf{Stat}(\mathit{sig})} \tag{9.68}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \mathit{sig}' := \left[\!\!\left[\begin{array}{l}\overline{\mathit{conid}}{\triangleright}\mathit{cvar}{:}\mathbf{T}, \\ \overline{\mathit{conid}}\_\mathrm{pack}{:}\mathit{sig}\xrightarrow{\mathsf{par}}[\![\mathrm{it}{:}\mathit{cvar}]\!], \\ \overline{\mathit{conid}}\_\mathrm{unpack}{:}[\![\mathrm{it}{:}\mathit{cvar}]\!]\xrightarrow{\mathsf{par}}\mathit{sig}\end{array}\right]\!\!\right]\end{array}}{\Gamma \vdash \texttt{packtype}\ \mathit{conid}\ \texttt{=}\ \mathit{sigexp} \leadsto \mathit{ilab}^*{:}\mathit{sig}'} \tag{9.69}$$

$$\frac{\mathit{sig}' := [\![\overline{\mathit{conid}}{:}\mathbf{T}, \overline{\mathit{conid}}\_\mathrm{pack}{:}[\![\cdot]\!], \overline{\mathit{conid}}\_\mathrm{unpack}{:}[\![\cdot]\!]]\!]}{\Gamma \vdash_{\mathrm{stat}} \texttt{packtype}\ \mathit{conid}\ \texttt{=}\ \mathit{sigexp} \leadsto \mathit{ilab}^*{:}\mathit{sig}'} \tag{9.70}$$

$$\frac{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \texttt{module}\ \mathit{modid}\ \texttt{:}\ \mathit{sigexp} \leadsto \overline{\mathit{modid}}{:}\mathit{sig}} \tag{9.71}$$

$$\frac{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig}}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \texttt{include}\ \mathit{sigexp} \leadsto \mathit{ilab}^*{:}\mathit{sig}} \tag{9.72}$$

Rule 9.72: The elaboration of `include` specs is analogous to the elaboration of `open` bindings.

$$\frac{\begin{array}{c}\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{specs} \leadsto \mathit{ldecs} \qquad \mathit{sig} := [\![\mathit{ldecs}]\!] \\ \Gamma, \mathit{mvar}{:}\mathit{sig}; \mathit{mvar}{:}\mathit{sig} \vdash_{\mathrm{sig}} \overline{\mathit{longconid}_1} \leadsto \mathit{mvar}^{\mathsf{c}}.\mathit{labs}_1 : \mathit{knd}_1 \\ \Gamma, \mathit{mvar}{:}\mathit{sig}; \mathit{mvar}{:}\mathit{sig} \vdash_{\mathrm{sig}} \overline{\mathit{longconid}_2} \leadsto \mathit{mvar}^{\mathsf{c}}.\mathit{labs}_2 : \mathit{knd}_2 \\ \left\{\begin{array}{c}\mathit{sig} \vdash_{\mathrm{sh}} \mathit{labs}_1 := \mathit{labs}_2 \leadsto \mathit{sig}' \\ \text{or} \\ \mathit{sig} \vdash_{\mathrm{sh}} \mathit{labs}_2 := \mathit{labs}_1 \leadsto \mathit{sig}'\end{array}\right\} \\ \mathit{sig}' = [\![\mathit{ldecs}']\!] \qquad \Gamma \vdash \mathit{ldecs}' \leq \mathit{ldecs}\end{array}}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{specs}\ \texttt{sharing type}\ \mathit{longconid}_1\ \texttt{=}\ \mathit{longconid}_2 \leadsto \mathit{ldecs}'} \tag{9.73}$$

Rule 9.73: We need the "or" choice in this rule because the symmetric `sharing` constraint does not indicate which of the two types is specified earlier in the *ldecs*.

$$\frac{\begin{array}{c}\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{specs} \leadsto \mathit{ldecs} \qquad \mathit{sig} := [\![\mathit{ldecs}]\!] \\ \Gamma, \mathit{mvar}{:}\mathit{sig}; \mathit{mvar}{:}\mathit{sig} \vdash_{\overline{\mathrm{sig}}} \overline{\mathit{longmodid}_1} \leadsto \mathit{pmod}_1 : \mathit{sig}_1 \\ \Gamma, \mathit{mvar}{:}\mathit{sig}; \mathit{mvar}{:}\mathit{sig} \vdash_{\overline{\mathrm{sig}}} \overline{\mathit{longmodid}_2} \leadsto \mathit{pmod}_2 : \mathit{sig}_2 \\ \mathsf{Fst}(\mathit{pmod}_1) = \mathit{mvar}^{\mathsf{c}}.\mathit{labs}_1 \qquad \mathsf{Fst}(\mathit{pmod}_2) = \mathit{mvar}^{\mathsf{c}}.\mathit{labs}_2 \\ \left\{\begin{array}{c} \mathit{sig} \vdash_{\overline{\mathrm{sh}}} \mathit{labs}_1 := \mathit{labs}_2 \leadsto \mathit{sig}' \\ \text{or} \\ \mathit{sig} \vdash_{\overline{\mathrm{sh}}} \mathit{labs}_2 := \mathit{labs}_1 \leadsto \mathit{sig}' \end{array}\right\} \\ \mathit{sig}' = [\![\mathit{ldecs}']\!] \qquad \Gamma \vdash \mathit{ldecs}' \leq \mathit{ldecs}\end{array}}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{specs}\ \mathtt{sharing}\ \mathit{longmodid}_1\ \mathtt{=}\ \mathit{longmodid}_2 \leadsto \mathit{ldecs}'} \tag{9.74}$$

Rule 9.74: We take Fst of pmod_1 and pmod_2 when computing labs_1 and labs_2 in order to eliminate any unroll's of rds's and just leave a sequence of projections.

Signature Expressions

$$\boxed{\begin{array}{r}\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \Gamma \vdash_{\overline{\mathrm{stat}}} \mathit{sigexp} \leadsto \mathit{statsig} : \mathsf{Sig}\end{array}}$$

$$\frac{\Gamma(\overline{\mathit{sigid}}) = \mathit{sig}}{\Gamma \vdash \mathit{sigid} \leadsto \mathit{sig} : \mathsf{Sig}} \tag{9.75}$$

$$\frac{\Gamma(\overline{\mathit{sigid}}) = \mathit{sig}}{\Gamma \vdash_{\overline{\mathrm{stat}}} \mathit{sigid} \leadsto \mathsf{Stat}(\mathit{sig}) : \mathsf{Sig}} \tag{9.76}$$

$$\frac{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{specs} \leadsto \mathit{ldecs}}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathtt{sig}\ \mathit{specs}\ \mathtt{end} \leadsto [\![\mathit{ldecs}]\!] : \mathsf{Sig}} \tag{9.77}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \Gamma, \overline{\mathit{modid}} \triangleright \mathit{mvar}{:}\mathit{sig} \vdash_{\langle\mathrm{stat}\rangle} \mathit{sigexp}' \leadsto \mathit{sig}' : \mathsf{Sig}\end{array}}{\Gamma \vdash_{\langle\mathrm{stat}\rangle} \mathtt{functor}\ (\mathit{modid}\ \mathtt{:}\ \mathit{sigexp})\ \mathtt{->}\ \mathit{sigexp}' \leadsto \Pi^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{sig}' : \mathsf{Sig}} \tag{9.78}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \Gamma, \overline{\mathit{modid}} \triangleright \mathit{mvar}{:}\mathit{sig} \vdash \mathit{sigexp}' \leadsto \mathit{sig}' : \mathsf{Sig}\end{array}}{\Gamma \vdash \mathtt{functor}\ (\mathit{modid}\ \mathtt{:}\ \mathit{sigexp})\ \mathtt{->>}\ \mathit{sigexp}' \leadsto \Pi^{\mathsf{par}}(\mathit{mvar}{:}\mathit{sig}).\mathit{sig}' : \mathsf{Sig}} \tag{9.79}$$

$$\frac{}{\Gamma \vdash_{\overline{\mathrm{stat}}} (\mathit{modid}\ \mathtt{:}\ \mathit{sigexp})\ \mathtt{->>}\ \mathit{sigexp}' \leadsto [\![\cdot]\!] : \mathsf{Sig}} \tag{9.80}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\overline{\mathrm{stat}}} \mathit{sigexp} \leadsto \mathit{statsig} : \mathsf{Sig} \\ \Gamma, \overline{\mathit{modid}} \triangleright \mathit{mvar}{:}\mathit{statsig} \vdash \mathit{sigexp} \leadsto \mathit{sig} : \mathsf{Sig} \\ \Gamma \vdash \mathit{statsig} \equiv \mathsf{Stat}(\mathit{sig})\end{array}}{\Gamma \vdash \mathtt{rec}\ (\mathit{modid})\ \mathit{sigexp} \leadsto \rho(\mathit{mvar}).\mathit{sig} : \mathsf{Sig}} \tag{9.81}$$

Rule 9.81: Note that the third premise is necessary because the static and normal elaborations of *sigexp* are performed under different contexts. In particular, the declaration of $\overline{\mathit{modid}}$ in the second premise may shadow earlier declarations of $\overline{\mathit{modid}}$ in the context Γ, in which case the first two premises alone are not enough to guarantee that the static part of *sigexp* does not refer to *modid*. (For instance, observe that if the third premise were omitted and a type `X.t` were visible in the context, then `rec (X) sig type t = X.t end` would be translated without complaint, and the output would be ill-formed.)

$$\frac{\Gamma \vdash_{\overline{\text{stat}}} \mathit{sigexp} \rightsquigarrow \mathit{statsig} : \mathsf{Sig}}{\Gamma \vdash_{\overline{\text{stat}}} \texttt{rec (}\mathit{modid}\texttt{)}\ \mathit{sigexp} \rightsquigarrow \rho(\mathit{statsig}) : \mathsf{Sig}} \tag{9.82}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\langle\text{stat}\rangle} \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig} \\ \Gamma, \overline{\mathit{tyvar}_1} \triangleright \mathit{cvar}_1{:}\mathbf{T}, \cdots, \overline{\mathit{tyvar}_n} \triangleright \mathit{cvar}_n{:}\mathbf{T} \vdash \mathit{ty} \rightsquigarrow \mathit{con} : \mathbf{T} \\ \Gamma, \mathit{mvar}{:}\mathit{sig}; \mathit{mvar}{:}\mathit{sig} \vdash_{\overline{\text{sig}}} \overline{\mathit{longconid}} \rightsquigarrow \mathit{mvar}^{\mathsf{c}}.\mathit{labs} : \mathbf{T}^n \rightarrow \mathbf{T} \\ \mathit{sig} \vdash_{\overline{\text{wt}}} \mathit{labs} := \lambda(\mathit{cvar}_1, \cdots, \mathit{cvar}_n).\mathit{con} \rightsquigarrow \mathit{sig}' \qquad \Gamma \vdash \mathit{sig}' \leq \mathit{sig}\end{array}}{\Gamma \vdash_{\langle\text{stat}\rangle} \mathit{sigexp}\ \texttt{where type (}\mathit{tyvar}_1\texttt{,} \cdots \texttt{,}\mathit{tyvar}_n\texttt{)}\ \mathit{longconid}\ \texttt{=}\ \mathit{ty} \rightsquigarrow \mathit{sig}' : \mathsf{Sig}} \tag{9.83}$$

Patterns

$$\boxed{\Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{else}\ \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}}$$

$$\frac{\text{Rules 9.88 and 9.90 do not apply.}}{\Gamma \vdash \mathit{expid} \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{else}\ \mathit{val} \rightsquigarrow \overline{\mathit{expid}}{=}\mathit{exp} : \overline{\mathit{expid}}{:}\mathit{con}} \tag{9.84}$$

Rule 9.84: Pattern match against an identifer (which is not a datatype or exception constructor) should always succeed.

$$\frac{\text{type}(\mathit{scon}) = \mathit{con}}{\begin{array}{l}\Gamma \vdash \mathit{scon} \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{else}\ \mathit{val} \rightsquigarrow \\ \qquad \mathit{ilab}{=}\mathsf{if}\ \mathit{basis}.\mathsf{eq}_{\mathit{con}}(\mathit{exp}, \mathit{scon})\ \mathsf{then}\ \{\}\ \mathsf{else}\ \mathsf{raise}^{\mathsf{Unit}}\ \mathit{val} : \mathit{ilab}{:}\mathsf{Unit}\end{array}} \tag{9.85}$$

Rule 9.85: Pattern match against a constant. We need primitive equality functions for constants appearing in patterns.

$$\frac{}{\Gamma \vdash \_ \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{else}\ \mathit{val} \rightsquigarrow \mathit{ilab}{=}\mathit{exp} : \mathit{ilab}{:}\mathit{con}} \tag{9.86}$$

Rule 9.86: Pattern match against a wildcard.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{ty} \rightsquigarrow \mathit{con}' : \mathbf{T} \qquad \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \mathbf{T} \\ \Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{else}\ \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}\end{array}}{\Gamma \vdash \mathit{pat}\ \texttt{:}\ \mathit{ty} \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{else}\ \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}} \tag{9.87}$$

Rule 9.87: Pattern match against an explicitly-typed pattern.

$$\frac{\begin{array}{c}\Gamma \vdash_{\overline{\text{ctx}}} \overline{\mathit{longexpid}} \rightsquigarrow \mathit{pmod}.\mathit{lab}_i : \forall(\mathit{cvar}_p{:}\mathit{knd}_p).\mathit{con} \qquad \mathit{con}' = \mathit{con}[\mathit{con}_p/\mathit{cvar}_p] \\ \Gamma \vdash \mathit{pmod}.\overline{\mathit{conid}}\_\mathrm{out} : \forall(\mathit{cvar}_p{:}\mathit{knd}_p).\mathit{con} \rightsquigarrow \mathit{con}^{\mathrm{sum}} \\ \mathit{con}^{\mathrm{sum}} = \Sigma\{\mathit{lab}_1{:}\mathit{con}_1, \ldots, \mathit{lab}_n{:}\mathit{con}_n\} \qquad \mathit{con}_i = \mathsf{Unit} \qquad \Gamma \vdash \mathit{con}_p : \mathit{knd}_p\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{longexpid} \Leftarrow \mathit{exp} : \mathit{con}'\ \mathsf{else}\ \mathit{val} \rightsquigarrow \\ \qquad \mathit{ilab}{=}\mathsf{pproj}_{\mathit{lab}_i}^{\mathit{con}^{\mathrm{sum}}[\mathit{con}_p/\mathit{cvar}_p]}(\mathit{pmod}.\overline{\mathit{conid}}\_\mathrm{out}[\mathit{con}_p]\langle\!\langle \mathit{exp}\rangle\!\rangle, \mathit{val}) : \mathit{ilab}{:}\mathsf{Unit}\end{array}} \tag{9.88}$$

Rule 9.88: Pattern match against a constant datatype constructor.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{\mathit{longexpid}} \rightsquigarrow \mathit{pmod}.\mathit{lab}_i : \forall(\mathit{cvar}_p{:}\mathit{knd}_p).\mathit{con}_i \to \mathit{con} \\ \mathit{con}_i' = \mathit{con}_i[\mathit{con}_p/\mathit{cvar}_p] \qquad \mathit{con}' = \mathit{con}[\mathit{con}_p/\mathit{cvar}_p] \\ \Gamma \vdash \mathit{pmod}.\overline{\mathit{conid}}\_\text{out} : \forall(\mathit{cvar}_p{:}\mathit{knd}_p).\mathit{con} \rightsquigarrow \mathit{con}^{\text{sum}} \\ \mathit{con}^{\text{sum}} = \Sigma\{\mathit{lab}_1{:}\mathit{con}_1, \ldots, \mathit{lab}_n{:}\mathit{con}_n\} \qquad \Gamma \vdash \mathit{con}_p : \mathit{knd}_p \\ \Gamma \vdash \mathit{pat} \Leftarrow \mathsf{pproj}^{\mathit{con}^{\text{sum}}[\mathit{con}_p/\mathit{cvar}_p]}_{\mathit{lab}_i}\,(\mathit{pmod}.\overline{\mathit{conid}}\_\text{out}[\mathit{con}_p]\langle\!\langle \mathit{exp}\rangle\!\rangle, \mathit{val}) : \mathit{con}_i' \;\mathsf{else}\; \mathit{val} \rightsquigarrow \\ \mathit{lbnds} : \mathit{ldecs}\end{array}}{\Gamma \vdash \mathit{longexpid}\ \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}' \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}} \tag{9.89}$$

Rule 9.89: Pattern match against a value-carrying datatype constructor.

$$\frac{\Gamma \vdash_{\text{ctx}} \overline{\mathit{longexpid}} \rightsquigarrow \mathit{pmod}.\mathit{lab} : \mathsf{Tagged} \qquad \Gamma \vdash \mathit{pmod}.\text{tag} : \mathsf{Unit\ Tag}}{\begin{array}{l}\Gamma \vdash \mathit{longexpid} \Leftarrow \mathit{exp} : \mathsf{Tagged} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \\ \quad \mathit{ilab}{=}\mathsf{iftagof}\ \mathit{exp}\ \mathsf{is}\ \mathit{pmod}.\text{tag}\ \mathsf{then}\ \lambda \mathit{evar}{:}\mathsf{Unit}.\{\}\ \mathsf{else}\ \mathsf{raise}^{\mathsf{Unit}}\ \mathit{val} : \mathit{ilab}{:}\mathsf{Unit}\end{array}} \tag{9.90}$$

Rule 9.90: Pattern match against a constant exception constructor.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{ctx}} \overline{\mathit{longexpid}} \rightsquigarrow \mathit{pmod}.\mathit{lab} : \mathit{con} \to \mathsf{Tagged} \\ \Gamma \vdash \mathit{pmod}.\text{tag} : \mathit{con}\ \mathsf{Tag} \\ \Gamma \vdash \mathit{pat} \Leftarrow (\mathsf{iftagof}\ \mathit{exp}\ \mathsf{is}\ \mathit{pmod}.\text{tag}\ \mathsf{then}\ \lambda \mathit{evar}{:}\mathit{con}.\mathit{evar}\ \mathsf{else}\ \mathsf{raise}^{\mathit{con}}\ \mathit{val}) : \mathit{con} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \\ \mathit{lbnds} : \mathit{ldecs}\end{array}}{\Gamma \vdash \mathit{longexpid}\ \mathit{pat} \Leftarrow \mathit{exp} : \mathsf{Tagged} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}} \tag{9.91}$$

Rule 9.91: Pattern match against a value-carrying exception constructor.

$$\frac{\begin{array}{c}\mathit{con} = \{\overline{\mathit{reclab}_1}{:}\mathit{con}_1, \cdots, \overline{\mathit{reclab}_n}{:}\mathit{con}_n\langle, \cdots\rangle\} \\ \forall i \in 1..n : \quad \Gamma \vdash \mathit{pat}_i \Leftarrow \pi_{\overline{\mathit{reclab}_i}}\,\mathit{exp} : \mathit{con}_i \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds}_i : \mathit{ldecs}_i\end{array}}{\begin{array}{l}\Gamma \vdash \{\mathit{reclab}_1 = \mathit{pat}_1, \cdots, \mathit{reclab}_n = \mathit{pat}_n\langle, \ldots\rangle\} \Leftarrow \mathit{exp} : \mathit{con} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \\ \quad \mathit{lbnds}_1, \cdots, \mathit{lbnds}_n : \mathit{ldecs}_1, \cdots, \mathit{ldecs}_n\end{array}} \tag{9.92}$$

Rule 9.92: Pattern match against a record of patterns. The syntactic concatenation of the $\mathit{lbnds}_i/\mathit{ldecs}_i$ is implicitly required to be well-formed. If it is not, it means that the pattern binds the same identifier twice, which is not permitted.

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{pat}_1 \Leftarrow \mathit{exp} : \mathit{con} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds}_1 : \mathit{ldecs}_1 \\ \Gamma \vdash \mathit{pat}_2 \Leftarrow \mathit{exp} : \mathit{con} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds}_2 : \mathit{ldecs}_2\end{array}}{\Gamma \vdash \mathit{pat}_1\ \mathtt{as}\ \mathit{pat}_2 \Leftarrow \mathit{exp} : \mathit{con} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds}_1, \mathit{lbnds}_2 : \mathit{ldecs}_1, \mathit{ldecs}_2} \tag{9.93}$$

Rule 9.93: Pattern match against two patterns simultaneously.

$$\frac{\Gamma \vdash \mathit{pat} \Leftarrow \mathsf{get}\ \mathit{exp} : \mathit{con} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}}{\Gamma \vdash \mathtt{ref}\ \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}\ \mathsf{Ref} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}} \tag{9.94}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con}' \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs} \\ \Gamma \vdash \mathit{con} \equiv \mathit{con}' : \mathbf{T}\end{array}}{\Gamma \vdash \mathit{pat} \Leftarrow \mathit{exp} : \mathit{con} \;\mathsf{else}\; \mathit{val} \rightsquigarrow \mathit{lbnds} : \mathit{ldecs}} \tag{9.95}$$

Datatype Definitions $\boxed{\Gamma \vdash \mathit{datbinds} \rightsquigarrow \mathit{sig}}$

$$
\frac{\begin{array}{l}
\forall i \in 1..p,\ \forall k \in 1..r : \\
\Gamma' := \Gamma, \mathit{ilab}^* \triangleright \mathit{mvar}^{\mathsf{c}}_{\mathrm{dt}} : \left[\!\!\left[\begin{array}{l} \mathit{ilab}^*_1 : [\![\overline{\mathit{conid}_1} : \mathbf{T}^{n_1} \to \mathbf{T}]\!], \cdots, \mathit{ilab}^*_p : [\![\overline{\mathit{conid}_p} : \mathbf{T}^{n_p} \to \mathbf{T}]\!], \\ \overline{\mathit{conid}'_1} : \mathbf{T}^{q_1} \to \mathbf{T}, \cdots, \overline{\mathit{conid}'_r} : \mathbf{T}^{q_r} \to \mathbf{T} \end{array} \right]\!\!\right] \\
\Gamma'_i := \Gamma', \overline{\mathit{tyvar}_{i1}} \triangleright \mathit{cvar}_{i1} : \mathbf{T}, \cdots, \overline{\mathit{tyvar}_{in_i}} \triangleright \mathit{cvar}_{in_i} : \mathbf{T} \\
\Gamma'' := \Gamma, \mathit{ilab}^*_1 \triangleright \mathit{mvar}^{\mathsf{c}}_1 : [\![\overline{\mathit{conid}_1} : \mathbf{T}^{n_1} \to \mathbf{T}]\!], \cdots, \mathit{ilab}^*_p \triangleright \mathit{mvar}^{\mathsf{c}}_p : [\![\overline{\mathit{conid}_p} : \mathbf{T}^{n_p} \to \mathbf{T}]\!] \\
\Gamma''_k := \Gamma'', \overline{\mathit{tyvar}'_{k1}} \triangleright \mathit{cvar}'_{k1} : \mathbf{T}, \cdots, \overline{\mathit{tyvar}'_{kq_k}} \triangleright \mathit{cvar}'_{kq_k} : \mathbf{T} \\
\Gamma'_i \vdash \left\{ \begin{array}{c} \mathtt{unit} \\ \text{or} \\ \mathit{ty}_{ij} \end{array} \right\}_{ij} \rightsquigarrow \mathit{con}_{ij} : \mathbf{T} \quad (\text{for } j \in 1..m_i) \qquad \Gamma''_k \vdash \mathit{ty}'_k \rightsquigarrow \mathit{con}'_k : \mathbf{T} \\
\mathit{con}^{\mathrm{sum}}_i := \Sigma\{ \overline{\mathit{expid}_{i1}} : \mathit{con}_{i1}, \cdots, \overline{\mathit{expid}_{im_i}} : \mathit{con}_{im_i} \} \\
\mathit{con}_i := (\mathit{mvar}^{\mathsf{c}}_{\mathrm{dt}}.\mathit{ilab}^*_i.\overline{\mathit{conid}_i})(\mathit{cvar}_{i1}, \cdots, \mathit{cvar}_{in_i}) \\
\mathit{sig}_i := \left[\!\!\left[\begin{array}{l}
\overline{\mathit{conid}_i} : \mathbf{T}^{n_i} \to \mathbf{T}, \\
\overline{\mathit{conid}_i}\_\mathrm{in} : \forall(\mathit{cvar}_{i1}, \cdots, \mathit{cvar}_{in_i}).\mathit{con}^{\mathrm{sum}}_i \rightsquigarrow \mathit{con}_i, \\
\overline{\mathit{conid}_i}\_\mathrm{out} : \forall(\mathit{cvar}_{i1}, \cdots, \mathit{cvar}_{in_i}).\mathit{con}_i \rightsquigarrow \mathit{con}^{\mathrm{sum}}_i, \\
\overline{\mathit{expid}_{ij}} : \forall(\mathit{cvar}_{i1}, \cdots, \mathit{cvar}_{in_i}).\left\{ \begin{array}{c} \mathit{con}_i \\ \text{or} \\ \mathit{con}_{ij} \to \mathit{con}_i \end{array} \right\}_{ij} \quad (\text{for } j \in 1..m_i)
\end{array} \right]\!\!\right] \\
\mathit{tknd}_k := \Pi(\mathit{cvar}'_{k1}, \cdots, \mathit{cvar}'_{kq_k}).\mathfrak{S}(\mathit{con}'_k) \\
\mathit{sig} := \rho(\mathit{mvar}_{\mathrm{dt}}).[\![\mathit{ilab}^*_1 : \mathit{sig}_1, \cdots, \mathit{ilab}^*_p : \mathit{sig}_p, \overline{\mathit{conid}'_1} : \mathit{tknd}_1, \cdots, \overline{\mathit{conid}'_r} : \mathit{tknd}_r]\!]
\end{array}}{\begin{array}{l}
\Gamma \vdash \mathtt{(}\mathit{tyvar}_{11}\mathtt{,} \cdots \mathtt{,}\mathit{tyvar}_{1n_1}\mathtt{)}\ \mathit{conid}_1\ \mathtt{=}\ \mathit{expid}_{11} \left\{ \begin{array}{c} \text{or} \\ \mathtt{of}\ \mathit{ty}_{11} \end{array} \right\}_{11} \mathtt{|} \cdots \mathtt{|}\ \mathit{expid}_{1m_1} \left\{ \begin{array}{c} \text{or} \\ \mathtt{of}\ \mathit{ty}_{1m_1} \end{array} \right\}_{1m_1} \rightsquigarrow \mathit{sig} \\
\quad \mathtt{and}\ \cdots\ \mathtt{and} \\
\quad \mathtt{(}\mathit{tyvar}_{p1}\mathtt{,} \cdots \mathtt{,}\mathit{tyvar}_{pn_p}\mathtt{)}\ \mathit{conid}_p\ \mathtt{=}\ \mathit{expid}_{p1} \left\{ \begin{array}{c} \text{or} \\ \mathtt{of}\ \mathit{ty}_{p1} \end{array} \right\}_{p1} \mathtt{|} \cdots \mathtt{|}\ \mathit{expid}_{pm_p} \left\{ \begin{array}{c} \text{or} \\ \mathtt{of}\ \mathit{ty}_{pm_p} \end{array} \right\}_{pm_p} \\
\quad \mathtt{withtype}\ \mathtt{(}\mathit{tyvar}'_{11}\mathtt{,} \cdots \mathtt{,}\mathit{tyvar}'_{1q_1}\mathtt{)}\ \mathit{conid}'_1\ \mathtt{=}\ \mathit{ty}'_1 \\
\quad \mathtt{and}\ \cdots\ \mathtt{and}\ \mathtt{(}\mathit{tyvar}'_{r1}\mathtt{,} \cdots \mathtt{,}\mathit{tyvar}'_{rq_r}\mathtt{)}\ \mathit{conid}'_r\ \mathtt{=}\ \mathit{ty}'_r
\end{array}} \tag{9.96}
$$

Rule 9.96: The elaboration of `datatype` definitions is actually quite straightforward; the rule looks large and complex mainly because there are a number of different n-ary entities:

- n_i = arity (number of type arguments) of the i^{th} `datatype` (out of p `datatype`'s)
- q_k = arity (number of type arguments) of the k^{th} `withtype` (out of r `withtype`'s)
- m_i = number of data constructors of the i^{th} `datatype`

Note that the `datatype`'s may refer to any of the `datatype`'s and any of the `withtype`'s, but the `withtype`'s may only refer to the `datatype`'s, not to one another.

Also note the use of an rds in the output signature to encode the dynamic-on-static recursive dependencies between the signatures of the `datatype` modules. Having rds's in the IL makes the encoding of `datatype`'s here somewhat simpler than it is in HS, where the corresponding rule is forced to fake an rds by "forward-declaring" the `datatype`'s.

9.3.3 Canonical Implementations of Signatures

$$\boxed{decs \vdash_{\text{can}} sig \rightsquigarrow mod}$$

$$\frac{\begin{array}{cc}\vdash_{\text{can}} \mathsf{Fst}(sig); \mathfrak{S}(cvar : sig) \rightsquigarrow con; mod & cvar \notin \text{dom}(decs) \\ mod' := \mathsf{let}\ cvar{=}\mu(cvar{:}\mathsf{Fst}(sig)).con\ \mathsf{in}\ (mod : sig) & decs \vdash mod' :_{\mathrm{P}} sig\end{array}}{decs \vdash_{\text{can}} sig \rightsquigarrow mod'} \tag{9.97}$$

Rule 9.97: *sig* is expected to only contain IL translations of `datatype` specs and *transparent* `type` specs (and module specs containing them). As outlined in Section 5.4, the basic idea here is to first compute a canonical implementation of $\mathsf{Fst}(sig)$ by joining all the type specs in *sig* together under one big μ-constructor. Once we have that μ-constructor, it is straightforward to define the canonical implementation of *sig*: type specifications are implemented by copying the corresponding components from the μ-constructor, and the only value specifications allowed in *sig* are the in and out coercions and data constructors for `datatype`'s, for which there are canonical implementations using fold's, unfold's and sum injections (see Rule 9.105).

The first premise invokes an auxiliary judgment that computes the canonical *con* of kind $\mathsf{Fst}(sig)$ and the canonical *mod* of signature *sig* simultaneously. (*con* cannot be computed just based on $\mathsf{Fst}(sig)$ because the value specifications in `datatype` specs in *sig* contain relevant type information that is not in $\mathsf{Fst}(sig)$.) The auxiliary judgment assumes for convenience that we use the same variable *cvar* both for the variable bound by the μ and the variable that the μ is bound to.

The last premise checks that the output module is well-formed, which is tantamount to checking that the kind $\mathsf{Fst}(sig)$ on the μ-constructor is expandable. See Section 6.2 for a discussion of the expandability restriction, and see Section 9.3.8 for more on the consequences of this restriction for recursive modules.

$$\boxed{\vdash_{\text{can}} lcdecs; ldecs \rightsquigarrow lcbnds; lbnds}$$

This and the following auxiliary judgments assume that the *sig* or *ldec*(*s*) input is in singleton form—specifically, that it has the form $\mathfrak{S}(cpath : sig)$ or $\mathfrak{S}(cpath : ldec(s))$, where *cpath* is a constructor path headed by the μ-variable *cvar* from Rule 9.97. Note that this precondition is indeed satisfied by the signature $\mathfrak{S}(cvar : sig)$ that is passed in originally in Rule 9.97. The singleton form assumption simplifies some of the rules, *e.g.,* the rds rule (Rule 9.104), which may assume the rds it is given is degenerate.

$$\frac{}{\vdash_{\text{can}} \cdot; \cdot \rightsquigarrow \cdot; \cdot} \tag{9.98}$$

$$\frac{\begin{array}{c}\vdash_{\text{can}} lcdec; ldec \rightsquigarrow lcbnd; lbnd \\ \vdash_{\text{can}} lcdecs; ldecs \rightsquigarrow lcbnds; lbnds\end{array}}{\vdash_{\text{can}} lcdec, lcdecs; ldec, ldecs \rightsquigarrow lcbnd, lcbnds; lbnd, lbnds} \tag{9.99}$$

$$\boxed{\vdash_{\text{can}} lcdec; ldec \rightsquigarrow lcbnd; lbnd}$$

$$\frac{}{\begin{array}{l}\vdash_{\text{can}} lab{\triangleright}cvar{:}tknd; lab{\triangleright}cvar{:}tknd' \rightsquigarrow \\ \qquad lab{\triangleright}cvar{=}\mathsf{Can}(tknd); lab{\triangleright}cvar{=}\mathsf{Can}(tknd')\end{array}} \tag{9.100}$$

Rule 9.100: For transparent type specifications, we can simply use the IL $\mathsf{Can}(\cdot)$ function.

$$\frac{\vdash_{\text{can}} \mathit{knd}; \mathit{sig} \rightsquigarrow \mathit{con}; \mathit{mod}}{\vdash_{\text{can}} \mathit{lab}{\triangleright}\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}; \mathit{lab}{\triangleright}\mathit{mvar}{:}\mathit{sig} \rightsquigarrow \mathit{lab}{\triangleright}\mathit{mvar}^{\mathsf{c}}{=}\mathit{con}; \mathit{lab}{\triangleright}\mathit{mvar}{=}\mathit{mod}} \tag{9.101}$$

$$\boxed{\vdash_{\text{can}} \mathit{knd}; \mathit{sig} \rightsquigarrow \mathit{con}; \mathit{mod}}$$

$$\frac{\vdash_{\text{can}} \mathit{lcdecs}; \mathit{ldecs} \rightsquigarrow \mathit{lcbnds}; \mathit{lbnds}}{\vdash_{\text{can}} [\![\mathit{lcdecs}]\!]; [\![\mathit{ldecs}]\!] \rightsquigarrow [\mathit{lcbnds}]; [\mathit{lbnds}]} \tag{9.102}$$

$$\frac{\vdash_{\text{can}} \mathit{knd}'; \mathit{sig}' \rightsquigarrow \mathit{con}; \mathit{mod}}{\begin{array}{c}\vdash_{\text{can}} \Pi(\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}).\mathit{knd}'; \Pi^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{sig}' \rightsquigarrow \\ \lambda(\mathit{mvar}^{\mathsf{c}}{:}\mathit{knd}).\mathit{con}; \lambda^{\mathsf{tot}}(\mathit{mvar}{:}\mathit{sig}).\mathit{mod}\end{array}} \tag{9.103}$$

$$\frac{\vdash_{\text{can}} \mathit{knd}; \mathit{sig} \rightsquigarrow \mathit{con}; \mathit{mod}}{\vdash_{\text{can}} \mathit{knd}; \rho(\mathit{sig}) \rightsquigarrow \mathit{con}; \mathsf{roll}(\mathit{mod})} \tag{9.104}$$

Rule 9.104: The rds is degenerate because it is assumed to be in singleton form.

$$\frac{\begin{array}{lcl}
\mathit{knd} & = & [\![\overline{\mathit{conid}}{:}\mathbf{T}^n{\to}\mathbf{T}]\!] \\
\mathit{con} & = & \mathit{cpath}.\overline{\mathit{conid}}(\mathit{cvar}_p) \\
\mathit{sig} & = & \left[\!\!\left[\begin{array}{l}
\overline{\mathit{conid}}{:}\Pi(\mathit{cvar}_p{:}\mathbf{T}^n).\mathfrak{S}(\mathit{con}), \\
\overline{\mathit{conid}}_\mathrm{in}{:}\forall(\mathit{cvar}_p{:}\mathbf{T}^n).\mathit{con}^{\mathrm{sum}} \rightsquigarrow \mathit{con}, \\
\overline{\mathit{conid}}_\mathrm{out}{:}\forall(\mathit{cvar}_p{:}\mathbf{T}^n).\mathit{con} \rightsquigarrow \mathit{con}^{\mathrm{sum}}, \\
\overline{\mathit{expid}_j}{:}\forall(\mathit{cvar}_p{:}\mathbf{T}^n).\left\{\begin{array}{c}\mathit{con} \\ \text{or} \\ \mathit{con}_j \to \mathit{con}\end{array}\right\}_j \quad (\text{for } j \in 1..m)
\end{array}\right]\!\!\right] \\
\mathit{con}' & := & [\overline{\mathit{conid}}{=}\lambda(\mathit{cvar}_p{:}\mathbf{T}^n).\mathit{con}^{\mathrm{sum}}] \\
\mathit{mod} & := & \left[\begin{array}{l}
\overline{\mathit{conid}}{=}\lambda(\mathit{cvar}_p{:}\mathbf{T}^n).\mathit{con}, \\
\overline{\mathit{conid}}_\mathrm{in}{=}\Lambda(\mathit{cvar}_p{:}\mathbf{T}^n).\mathsf{fold}^{\mathit{con}}, \\
\overline{\mathit{conid}}_\mathrm{out}{=}\Lambda(\mathit{cvar}_p{:}\mathbf{T}^n).\mathsf{unfold}^{\mathit{con}}, \\
\overline{\mathit{expid}_j}{=}\Lambda(\mathit{cvar}_p{:}\mathbf{T}^n).\left\{\begin{array}{c}\mathsf{fold}^{\mathit{con}}\langle\!\langle \mathsf{inj}^{\mathit{con}^{\mathrm{sum}}}_{\mathit{expid}_j}\ \{\}\rangle\!\rangle \\ \text{or} \\ \lambda(\mathit{evar}{:}\mathit{con}_j).\mathsf{fold}^{\mathit{con}}\langle\!\langle \mathsf{inj}^{\mathit{con}^{\mathrm{sum}}}_{\mathit{expid}_j}\ \mathit{evar}\rangle\!\rangle\end{array}\right\}_j
\end{array}\right]
\end{array}}{\vdash_{\text{can}} \mathit{knd}; \mathit{sig} \rightsquigarrow \mathit{con}'; \mathit{mod}} \tag{9.105}$$

Rule 9.105: The canonical implementation of a `datatype` spec. A few points of note:

- We can assume that *sig* specifies the constructor $\overline{\mathit{conid}}$ as transparently equal to a type of the form $\mathit{cpath}.\overline{\mathit{conid}}$ because *sig* is assumed to be of the form $\mathfrak{S}(\mathit{cpath} : \mathit{sig}')$.
- This rule illustrates that it is necessary to have *sig* around in order to compute the output con', because the definition of con' comes from looking at the types of $\overline{\mathit{conid}}$'s in and out coercions, which do not show up in the input *knd*.
- con' implements $\overline{\mathit{conid}}$ as just a sum type, not a *recursive* sum type; all the recursive knot-tying will be handled by the μ that Rule 9.97 wraps around the whole constructor at the very end.

9.3.4 Coercive Signature Matching

$$\boxed{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq ldecs \rightsquigarrow plbnds : tldecs}$$

$$\frac{}{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq \cdot \rightsquigarrow \cdot : \cdot} \tag{9.106}$$

$$\frac{\begin{array}{c} decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq ldec \rightsquigarrow plbnd : tldec \\ decs, tldec \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq ldecs \rightsquigarrow plbnds : tldecs \end{array}}{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq ldec, ldecs \rightsquigarrow plbnd, plbnds : tldec, tldecs} \tag{9.107}$$

$$\boxed{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq ldec \rightsquigarrow plbnd : tldec}$$

$$\frac{\begin{array}{c} decs; vpmod{:}sig_0 \vdash_{\overline{\text{sig}}} lab \rightsquigarrow vexp : \forall(cvar'_p{:}knd'_p).con' \\ decs, cvar_p{:}knd_p \vdash con'_p : knd'_p \qquad decs, cvar_p{:}knd_p \vdash con \equiv con'[con'_p/cvar'_p] : \mathbf{T} \end{array}}{\begin{array}{l} decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq lab{\triangleright}evar{:}\forall(cvar_p{:}knd_p).con \rightsquigarrow \\ \qquad lab{\triangleright}evar{=}\Lambda(cvar_p{:}knd_p).(vexp[con'_p]) : lab{\triangleright}evar{:}\forall(cvar_p{:}knd_p).con \end{array}} \tag{9.108}$$

Rule 9.108: Coercion to a (potentially) polymorphic value specification; this may involve implicit polymorphic instantiation. Note that this rule also handles alpha-conversion of EL type variables, as knd_p and knd'_p need not have the same labels on their type components.

$$\frac{decs; vpmod{:}sig_0 \vdash_{\overline{\text{sig}}} lab \rightsquigarrow con : knd}{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq lab{\triangleright}cvar{:}knd \rightsquigarrow lab{\triangleright}cvar{=}con : lab{\triangleright}cvar{:}\mathfrak{S}(con : knd)} \tag{9.109}$$

Rule 9.109: Coercion to a type constructor specification.

$$\frac{\begin{array}{c} lab \text{ is not open} \\ decs; vpmod{:}sig_0 \vdash_{\overline{\text{sig}}} lab \rightsquigarrow vpmod' : sig' \\ decs \vdash_{\overline{\text{sub}}} vpmod' : sig' \preceq sig \rightsquigarrow pmod : tsig \end{array}}{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq lab{\triangleright}mvar{:}sig \rightsquigarrow lab{\triangleright}mvar{=}pmod : lab{\triangleright}mvar{:}tsig} \tag{9.110}$$

Rule 9.110: Coercion to a module specification.

$$\frac{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq sig \rightsquigarrow pmod : tsig}{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq lab^*{\triangleright}mvar{:}sig \rightsquigarrow lab^*{\triangleright}mvar{=}pmod : lab^*{\triangleright}mvar{:}tsig} \tag{9.111}$$

Rule 9.111: Coercion to an open module specification, which means that *vpmod* need not provide a component labeled lab^*, but it must provide all the components in the signature *sig*.

$$\boxed{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq sig \rightsquigarrow pmod : tsig}$$

$$\frac{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq ldecs \rightsquigarrow plbnds : tldecs}{decs \vdash_{\overline{\text{sub}}} vpmod : sig_0 \preceq [\![ldecs]\!] \rightsquigarrow [plbnds] : [\![tldecs]\!]} \tag{9.112}$$

$$\frac{\begin{array}{c} decs \vdash_{\text{peel}} vpmod_0{:}sig_0 \rightsquigarrow vpmod : \Pi^{\mathsf{tot}}(mvar_1{:}sig_1).sig'_1 \\ decs, mvar_2{:}sig_2 \vdash_{\text{sub}} mvar_2 : sig_2 \preceq sig_1 \rightsquigarrow pmod_1 : \_ \\ decs, mvar_2{:}sig_2, mvar'_1{:}sig'_1[pmod_1/mvar_1] \vdash_{\text{sub}} \\ \quad mvar'_1 : sig'_1[pmod_1/mvar_1] \preceq sig'_2 \rightsquigarrow pmod'_2 : tsig'_2 \end{array}}{\begin{array}{l} decs \vdash_{\text{sub}} vpmod_0 : sig_0 \preceq \Pi^{\mathsf{tot}}(mvar_2{:}sig_2).sig'_2 \rightsquigarrow \\ \quad \lambda^{\mathsf{tot}}(mvar_2{:}sig_2).\mathsf{plet}\ mvar'_1{=}vpmod^{\mathsf{tot}}(pmod_1)\ \mathsf{in}\ pmod'_2 : \\ \quad \Pi^{\mathsf{tot}}(mvar_2{:}sig_2).tsig'_2[vpmod^{\mathsf{tot}}(pmod_1)/mvar'_1] \end{array}} \tag{9.113}$$

Rule 9.113: Coercion to a total (applicative) functor specification.

$$\frac{\begin{array}{c} decs \vdash_{\text{peel}} vpmod_0{:}sig_0 \rightsquigarrow vpmod : \Pi^{\tau}(mvar_1{:}sig_1).sig'_1 \\ decs, mvar_2{:}sig_2 \vdash_{\text{sub}} mvar_2 : sig_2 \preceq sig_1 \rightsquigarrow pmod_1 : \_ \\ decs, mvar_2{:}sig_2, mvar'_1{:}sig'_1[pmod_1/mvar_1] \vdash_{\text{sub}} \\ \quad mvar'_1 : sig'_1[pmod_1/mvar_1] \preceq sig'_2 \rightsquigarrow pmod'_2 : \_ \end{array}}{\begin{array}{l} decs \vdash_{\text{sub}} vpmod_0 : sig_0 \preceq \Pi^{\mathsf{par}}(mvar_2{:}sig_2).sig'_2 \rightsquigarrow \\ \quad \lambda^{\mathsf{par}}(mvar_2{:}sig_2){:}sig'_2.\mathsf{let}\ mvar'_1{=}vpmod^{\tau}(pmod_1)\ \mathsf{in}\ (pmod'_2 : sig'_2) : \\ \quad \Pi^{\mathsf{par}}(mvar_2{:}sig_2).sig'_2 \end{array}} \tag{9.114}$$

Rule 9.114: Coercion to a partial (generative) functor specification.

$$\frac{\begin{array}{c} decs, mvar_{\text{stat}}{:}\mathsf{Stat}(sig_0) \vdash_{\text{sub}} mvar_{\text{stat}} : \mathsf{Stat}(sig_0) \preceq \mathsf{Stat}(sig) \rightsquigarrow pmod_{\text{stat}} : \_ \\ con := \mathsf{Fst}(pmod_{\text{stat}})[\mathsf{Fst}(vpmod)/mvar^{\mathsf{c}}_{\text{stat}}] \\ decs \vdash_{\text{sub}} vpmod : sig_0 \preceq \mathfrak{S}(con : sig[con/mvar^{\mathsf{c}}]) \rightsquigarrow pmod : tsig \end{array}}{decs \vdash_{\text{sub}} vpmod : sig_0 \preceq \rho(mvar).sig \rightsquigarrow \mathsf{roll}(pmod) : \rho(tsig)} \tag{9.115}$$

Rule 9.115: Coercion to a recursively dependent signature. The first premise serves essentially as a way of coercing $\mathsf{Fst}(sig_0)$ to the *kind* $\mathsf{Fst}(sig)$. Instead of defining a whole other judgment of coercive kind matching, we simply coerce $\mathsf{Stat}(sig_0)$ to $\mathsf{Stat}(sig)$ and then take Fst of the coercion module. (The only slightly tricky point is that $\mathsf{Stat}(\cdot)$ is not defined for modules, so we must make up a module variable $mvar_{\text{stat}}$ and then substitute $\mathsf{Fst}(vpmod)$ for it later on.) This is a canonical example of where static signatures and the $\mathsf{Stat}(\cdot)$ function come in handy.

Once we have a constructor *con* of kind $\mathsf{Fst}(sig)$, we can coerce to the body of the rds (substituting *con* in place of the recursive variable) and then roll the coercion module into the rds.

9.3.5 Signature Patching

where type $\boxed{sig \vdash_{\text{wt}} labs := phrase \rightsquigarrow sig'}$

$$\frac{\begin{array}{c} \mathrm{FV}(phrase) \cap \mathrm{vardom}(ldecs) = \emptyset \\ sig = [\![ldecs, lab{\triangleright}var{:}class, ldecs']\!] \end{array}}{sig \vdash_{\text{wt}} lab := phrase \rightsquigarrow [\![ldecs, lab{\triangleright}var{:}\mathfrak{S}(phrase : class), ldecs']\!]} \tag{9.116}$$

$$\frac{\begin{array}{c} \mathrm{FV}(phrase) \cap \mathrm{vardom}(ldecs) = \emptyset \\ sig = [\![ldecs, lab{\triangleright}mvar{:}sig', ldecs']\!] \\ sig' \vdash_{\text{wt}} labs := phrase \rightsquigarrow sig'' \end{array}}{sig \vdash_{\text{wt}} lab.labs := phrase \rightsquigarrow [\![ldecs, lab{\triangleright}mvar{:}sig'', ldecs']\!]} \tag{9.117}$$

$$\frac{mvar \notin \mathrm{FV}(phrase) \qquad sig = \rho(mvar).sig' \qquad sig' \vdash_{\mathrm{wt}} labs := phrase \rightsquigarrow sig''}{sig \vdash_{\mathrm{wt}} labs := phrase \rightsquigarrow \rho(mvar).sig''} \tag{9.118}$$

sharing $\boxed{sig \vdash_{\mathrm{sh}} labs_1 := labs_2 \rightsquigarrow sig'}$

$$\frac{sig = [\![ldecs, lab'\triangleright var'{:}class', ldecs', lab\triangleright var{:}class, ldecs'']\!]}{sig \vdash_{\mathrm{sh}} lab := lab' \rightsquigarrow [\![ldecs, lab'\triangleright var'{:}class', ldecs', lab\triangleright var{:}\mathfrak{S}(var' : class), ldecs'']\!]} \tag{9.119}$$

$$\frac{\begin{array}{c} sig = [\![ldecs, lab'\triangleright var'{:}class', ldecs', lab\triangleright var{:}sig'', ldecs'']\!] \\ sig'' \vdash_{\mathrm{wt}} labs := var' \rightsquigarrow sig''' \end{array}}{sig \vdash_{\mathrm{sh}} lab.labs := lab' \rightsquigarrow [\![ldecs, lab'\triangleright var'{:}class', ldecs', lab\triangleright var{:}sig''', ldecs'']\!]} \tag{9.120}$$

$$\frac{\begin{array}{c} sig = [\![ldecs, lab'\triangleright var'{:}sig', ldecs', lab\triangleright var{:}sig'', ldecs'']\!] \\ sig'' \vdash_{\mathrm{wt}} labs := var'.labs' \rightsquigarrow sig''' \end{array}}{sig \vdash_{\mathrm{sh}} lab.labs := lab'.labs' \rightsquigarrow [\![ldecs, lab'\triangleright var'{:}sig', ldecs', lab\triangleright var{:}sig''', ldecs'']\!]} \tag{9.121}$$

$$\frac{sig = [\![ldecs, lab\triangleright var{:}sig', ldecs']\!] \qquad sig' \vdash_{\mathrm{sh}} labs := labs' \rightsquigarrow sig''}{sig \vdash_{\mathrm{sh}} lab.labs := lab.labs' \rightsquigarrow [\![ldecs, lab\triangleright var{:}sig'', ldecs']\!]} \tag{9.122}$$

$$\frac{sig = \rho(mvar).sig' \qquad sig' \vdash_{\mathrm{sh}} labs := labs' \rightsquigarrow sig''}{sig \vdash_{\mathrm{sh}} labs := labs' \rightsquigarrow \rho(mvar).sig''} \tag{9.123}$$

9.3.6 Signature Peeling

$\boxed{decs \vdash_{\mathrm{peel}} pmod{:}sig \rightsquigarrow pmod' : sig'}$

$$\frac{\vdash_{\mathrm{peel}} pmod{:}sig \rightsquigarrow pmod' \qquad decs \vdash pmod' :_{\mathrm{P}} sig'}{decs \vdash_{\mathrm{peel}} pmod{:}sig \rightsquigarrow pmod' : sig'} \tag{9.124}$$

$\boxed{\vdash_{\mathrm{peel}} pmod{:}sig \rightsquigarrow pmod'}$

Auxiliary peeling judgment, which strips off all leading existentials, rds's and maybe's.

$$\frac{\vdash_{\mathrm{peel}} pmod.\mathrm{visible}^*{:}sig_2 \rightsquigarrow pmod'}{\vdash_{\mathrm{peel}} pmod{:}\exists(mvar{:}sig_1).sig_2 \rightsquigarrow pmod'} \tag{9.125}$$

$$\frac{\vdash_{\mathrm{peel}} \mathsf{unroll}(pmod){:}sig \rightsquigarrow pmod'}{\vdash_{\mathrm{peel}} pmod{:}\rho(mvar).sig \rightsquigarrow pmod'} \tag{9.126}$$

$$\frac{\vdash_{\mathrm{peel}} \mathsf{fetch}(pmod){:}sig \rightsquigarrow pmod'}{\vdash_{\mathrm{peel}} pmod{:}\mathsf{maybe}(sig) \rightsquigarrow pmod'} \tag{9.127}$$

$$\frac{\text{The above rules do not apply.}}{\vdash_{\mathrm{peel}} pmod{:}sig \rightsquigarrow pmod} \tag{9.128}$$

9.3.7 Label Lookup

Context Lookup $\boxed{\Gamma \vdash_{\mathrm{ctx}} labs \leadsto phrase : class}$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} labs \leadsto con \qquad \Gamma \vdash con : knd}{\Gamma \vdash_{\mathrm{ctx}} labs \leadsto con : knd} \tag{9.129}$$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} labs \leadsto exp \qquad \Gamma \vdash exp : con}{\Gamma \vdash_{\mathrm{ctx}} labs \leadsto exp : con} \tag{9.130}$$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} labs \leadsto pmod \qquad \Gamma \vdash pmod :_{\mathrm{P}} sig}{\Gamma \vdash_{\mathrm{ctx}} labs \leadsto pmod : sig} \tag{9.131}$$

$\boxed{\Gamma \vdash_{\mathrm{ctx}} labs \leadsto phrase}$

Auxiliary context lookup judgment.

$$\frac{}{\Gamma, lab \triangleright cvar{:}knd \vdash_{\mathrm{ctx}} lab \leadsto cvar} \tag{9.132}$$

$$\frac{lab \neq lab' \qquad \Gamma \vdash_{\mathrm{ctx}} lab' \leadsto phrase}{\Gamma, lab \triangleright cvar{:}knd \vdash_{\mathrm{ctx}} lab' \leadsto phrase} \tag{9.133}$$

$$\frac{}{\Gamma, lab \triangleright evar{:}con \vdash_{\mathrm{ctx}} lab \leadsto evar} \tag{9.134}$$

$$\frac{lab \neq lab' \qquad \Gamma \vdash_{\mathrm{ctx}} lab' \leadsto phrase}{\Gamma, lab \triangleright evar{:}con \vdash_{\mathrm{ctx}} lab' \leadsto phrase} \tag{9.135}$$

$$\frac{\vdash_{\mathrm{peel}} mvar{:}sig \leadsto pmod \qquad lab \text{ is not open}}{\Gamma, lab \triangleright mvar{:}sig \vdash_{\mathrm{ctx}} lab \leadsto pmod} \tag{9.136}$$

Rule 9.136: If we find the label and it corresponds to a module, we return the module in peeled form.

$$\frac{lab \neq lab' \qquad \Gamma \vdash_{\mathrm{ctx}} lab' \leadsto phrase \qquad lab \text{ is not open}}{\Gamma, lab \triangleright mvar{:}sig \vdash_{\mathrm{ctx}} lab' \leadsto phrase} \tag{9.137}$$

$$\frac{mvar{:}sig \vdash_{\mathrm{sig}} lab' \leadsto phrase}{\Gamma, lab^{*} \triangleright mvar{:}sig \vdash_{\mathrm{ctx}} lab' \leadsto phrase} \tag{9.138}$$

Rule 9.138: When we hit an open label in the context, we switch to signature lookup in order to search for lab' inside sig.

$$\frac{mvar{:}sig \vdash_{\mathrm{sig}} lab' \not\leadsto \qquad \Gamma \vdash_{\mathrm{ctx}} lab' \leadsto phrase}{\Gamma, lab^{*} \triangleright mvar{:}sig \vdash_{\mathrm{ctx}} lab' \leadsto phrase} \tag{9.139}$$

$$\frac{\Gamma \vdash_{\mathrm{ctx}} lab \leadsto pmod : sig \qquad \Gamma; pmod{:}sig \vdash_{\mathrm{sig}} labs \leadsto phrase : class}{\Gamma \vdash_{\mathrm{ctx}} lab.labs \leadsto phrase} \tag{9.140}$$

Signature Lookup $\boxed{decs; pmod{:}sig \vdash_{\overline{\text{sig}}} labs \leadsto phrase : class}$

$$\frac{decs \vdash pmod :_{\mathrm{P}} sig \qquad pmod{:}sig \vdash_{\overline{\text{sig}}} lab \leadsto phrase \qquad decs \vdash phrase : class}{decs; pmod{:}sig \vdash_{\overline{\text{sig}}} lab \leadsto phrase : class} \tag{9.141}$$

$$\frac{\begin{array}{c} decs; pmod{:}sig \vdash_{\overline{\text{sig}}} lab \leadsto pmod' : sig' \\ decs; pmod'{:}sig' \vdash_{\overline{\text{sig}}} labs \leadsto phrase : class \end{array}}{decs; pmod{:}sig \vdash_{\overline{\text{sig}}} lab.labs \leadsto phrase : class} \tag{9.142}$$

$\boxed{pmod{:}sig \vdash_{\overline{\text{sig}}} lab \leadsto phrase}$

Auxiliary signature lookup judgment, defined very similarly to the auxiliary context lookup judgment.

$$\frac{}{pmod{:}[\![ldecs, lab{\triangleright}cvar{:}knd]\!] \vdash_{\overline{\text{sig}}} lab \leadsto \mathsf{Fst}(pmod).lab} \tag{9.143}$$

$$\frac{lab \neq lab' \qquad pmod{:}[\![ldecs]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase}{pmod{:}[\![ldecs, lab{\triangleright}cvar{:}knd]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase} \tag{9.144}$$

$$\frac{}{pmod{:}[\![ldecs, lab{\triangleright}evar{:}con]\!] \vdash_{\overline{\text{sig}}} lab \leadsto pmod.lab} \tag{9.145}$$

$$\frac{lab \neq lab' \qquad pmod{:}[\![ldecs]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase}{pmod{:}[\![ldecs, lab{\triangleright}evar{:}con]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase} \tag{9.146}$$

$$\frac{\vdash_{\overline{\text{peel}}} pmod.lab{:}sig \leadsto phrase \qquad lab \text{ is not open}}{pmod{:}[\![ldecs, lab{\triangleright}mvar{:}sig]\!] \vdash_{\overline{\text{sig}}} lab \leadsto phrase} \tag{9.147}$$

$$\frac{lab \neq lab' \qquad pmod{:}[\![ldecs]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase \qquad lab \text{ is not open}}{pmod{:}[\![ldecs, lab{\triangleright}mvar{:}sig]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase} \tag{9.148}$$

$$\frac{pmod.lab^*{:}sig \vdash_{\overline{\text{sig}}} lab' \leadsto phrase}{pmod{:}[\![ldecs, lab^*{\triangleright}mvar{:}sig]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase} \tag{9.149}$$

$$\frac{pmod.lab^*{:}sig \vdash_{\overline{\text{sig}}} lab' \not\leadsto \qquad pmod{:}[\![ldecs]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase}{pmod{:}[\![ldecs, lab^*{\triangleright}mvar{:}sig]\!] \vdash_{\overline{\text{sig}}} lab' \leadsto phrase} \tag{9.150}$$

$$\frac{\mathsf{unroll}(pmod){:}sig \vdash_{\overline{\text{sig}}} lab \leadsto phrase}{pmod{:}\rho(mvar).sig \vdash_{\overline{\text{sig}}} lab \leadsto phrase} \tag{9.151}$$

9.3.8 Recursive Module Elaboration

I will begin by giving the elaboration rule for recursive module expressions. As described in Section 5.4, this rule elaborates the recursive module body in two phases. The first "static" phase elaborates only the static components of the module and produces a meta-signature. In the second "main" phase, the meta-signature is used to adjust the amount of type information revealed in the typing context at different points during the typechecking of the module body.

$$\boxed{\Gamma \vdash \mathit{modexp} \rightsquigarrow \mathit{mod} :_{\kappa} \mathit{sig}}$$

$$\frac{\begin{array}{l}
1.\ \Gamma \vdash \overline{\mathit{sigexp}} \rightsquigarrow \mathit{sig}_{\mathrm{rec}} : \mathsf{Sig} \\
2.\ \Gamma, \overline{\mathit{modid}}{\triangleright}\mathit{mvar}_{\mathrm{rec}}{:}\mathit{sig}_{\mathrm{rec}} \vdash_{\mathrm{stat}} \mathit{modexp} \rightsquigarrow \mathit{metasig}_{\mathrm{actual}} \\
3.\ \mathit{mvar}^{\mathsf{c}}_{\mathrm{rec}} \notin \mathrm{FV}(\mathsf{Stat}(\mathsf{Priv}(\mathit{metasig}_{\mathrm{actual}}))) \cup \mathrm{FV}(\mathsf{Stat}(\mathsf{Pub}(\mathit{metasig}_{\mathrm{actual}}))) \\
4.\ \Gamma, \mathit{mvar}_{\mathrm{actual}}{:}\mathsf{Stat}(\mathsf{Pub}(\mathit{metasig}_{\mathrm{actual}})) \vdash_{\mathrm{sub}} \\
\qquad \mathit{mvar}_{\mathrm{actual}} : \mathsf{Stat}(\mathsf{Pub}(\mathit{metasig}_{\mathrm{actual}})) \preceq \mathsf{Stat}(\mathit{sig}_{\mathrm{rec}}) \rightsquigarrow _ : \mathit{tstatsig}_{\mathrm{coerced}} \\
5.\ \mathit{metasig}'_{\mathrm{actual}} := \mathit{metasig}_{\mathrm{actual}}[\mathit{mvar}^{\mathsf{c}}_{\mathrm{rec}}.\mathrm{visible}^{*}/\mathit{mvar}^{\mathsf{c}}_{\mathrm{rec}}] \\
6.\ \mathit{metasig}_{\mathrm{static}} := \rho(\mathit{mvar}_{\mathrm{rec}}).\exists(\mathit{mvar}_{\mathrm{actual}}{:}\boxed{\mathit{metasig}'_{\mathrm{actual}}}).\mathit{tstatsig}_{\mathrm{coerced}} \\
7.\ \Gamma \vdash_{\mathrm{can}} \mathsf{Priv}(\mathit{metasig}_{\mathrm{static}}) \rightsquigarrow \mathit{mod}_{\mathrm{static}} \\
8.\ \mathit{tsig}_{\mathrm{rec}} := \mathfrak{S}(\mathit{mvar}^{\mathsf{c}}_{\mathrm{static}}.\mathrm{visible}^{*} : \mathit{sig}_{\mathrm{rec}}) \\
9.\ \Theta := \Gamma; \mathit{mvar}_{\mathrm{static}}{:}\mathit{metasig}_{\mathrm{static}}; \overline{\mathit{modid}}{\triangleright}\mathit{mvar}_{\mathrm{rec}}{:}\mathsf{maybe}(\mathit{tsig}_{\mathrm{rec}}) \\
10.\ \Theta \vdash_{\mathrm{rec}} \mathsf{unroll}(\mathit{mvar}_{\mathrm{static}}).\mathrm{hidden} \Rightarrow \mathit{modexp} \rightsquigarrow \mathit{mod}_{\mathrm{actual}} : \mathit{tsig}_{\mathrm{actual}} \\
11.\ \Theta, \mathit{mvar}_{\mathrm{actual}}{:}\mathit{tsig}_{\mathrm{actual}} \vdash_{\mathrm{sub}} \mathit{mvar}_{\mathrm{actual}} : \mathit{tsig}_{\mathrm{actual}} \preceq \mathit{tsig}_{\mathrm{rec}} \rightsquigarrow \mathit{pmod}_{\mathrm{coerced}} : _ \\
12.\ \mathit{mod} := \mathsf{let}\ \mathit{mvar}_{\mathrm{static}}{=}\mathit{mod}_{\mathrm{static}}\ \mathsf{in} \\
\qquad\qquad (\mathsf{rec}(\mathit{mvar}_{\mathrm{rec}}{:}\mathit{tsig}_{\mathrm{rec}}.\mathsf{let}\ \mathit{mvar}_{\mathrm{actual}}{=}\mathit{mod}_{\mathrm{actual}}\ \mathsf{in}\ (\mathit{pmod}_{\mathrm{coerced}} : \mathit{tsig}_{\mathrm{rec}})) \\
\qquad\qquad\quad : \mathit{sig}_{\mathrm{rec}})
\end{array}}{\Gamma \vdash \texttt{rec (}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \mathit{modexp} \rightsquigarrow \mathit{mod} :_{\mathsf{P}} \mathit{sig}_{\mathrm{rec}}} \quad (9.152)$$

Rule 9.152: Let us consider the premises one at a time. (I suggest the reader compare the formal steps here with the high-level description given in Section 5.4, as they correspond quite closely.)

1. Translate the declared signature *sigexp* to $\mathit{sig}_{\mathrm{rec}}$, which need not be transparent.

2. Perform static elaboration of *modexp*, resulting in the meta-signature $\mathit{metasig}_{\mathrm{actual}}$.

3. Enforce the "dynamic-on-static" restriction on *modexp* by checking that the recursive module variable $\mathit{mvar}_{\mathrm{rec}}$ does not appear in the static part of $\mathit{metasig}_{\mathrm{actual}}$. References to $\mathit{mvar}_{\mathrm{rec}}$ may still occur in $\mathit{metasig}_{\mathrm{actual}}$'s value specs (*i.e.,* its `datatype` specs).

4. We need to close up references to $\mathit{mvar}_{\mathrm{rec}}$ in $\mathit{metasig}_{\mathrm{actual}}$ by enclosing it in an rds (cf. Figure 5.17). Unfortunately, we cannot just write $\rho(\mathit{mvar}_{\mathrm{rec}}).\mathit{metasig}_{\mathrm{actual}}$, because (by step 2) $\mathit{metasig}_{\mathrm{actual}}$ expects $\mathit{mvar}^{\mathsf{c}}_{\mathrm{rec}}$ to have kind $\mathsf{Fst}(\mathit{sig}_{\mathrm{rec}})$, not $\mathsf{Fst}(\mathit{metasig}_{\mathrm{actual}})$. So first we coerce $\mathit{metasig}_{\mathrm{actual}}$ into the shape of $\mathsf{Stat}(\mathit{sig}_{\mathrm{rec}})$, which produces $\mathit{tstatsig}_{\mathrm{coerced}}$.

5–6. Using $\mathit{tstatsig}_{\mathrm{coerced}}$, we can close up $\mathit{metasig}_{\mathrm{actual}}$ with an rds and call it $\mathit{metasig}_{\mathrm{static}}$ (in Section 5.4, this was called `CSS`). For now, ignore the box around $\mathit{metasig}'_{\mathrm{actual}}$. The purpose of the box will be explained below when the $\vdash_{\mathrm{rec}}$ judgment is defined.

7. Construct the canonical module $\mathit{mod}_{\mathrm{static}}$ matching $\mathit{metasig}_{\mathrm{static}}$. Note: this will only succeed if $\mathsf{Fst}(\mathsf{Priv}(\mathit{metasig}_{\mathrm{static}}))$ is an expandable kind. From a programming perspective, this means that if in *modexp* there appears a total `functor` expression whose body contains `datatype` definitions, then the argument signature of that functor must not contain any transparent type specifications. I admit this is a rather bizarre restriction, but I see no way around it.

8. $\mathit{mod}_{\mathrm{static}}$ will eventually be bound to $\mathit{mvar}_{\mathrm{static}}$ (in Section 5.4, this was called `Static`). We can thus selfify the declared signature $\mathit{sig}_{\mathrm{rec}}$ with respect to $\mathit{mvar}_{\mathrm{static}}.\mathrm{visible}^{*}$, in order to obtain a *transparent* version of the declared signature $\mathit{tsig}_{\mathrm{rec}}$.

9. The typing context Θ for the main phase of elaboration binds the recursive module variable with the signature $\mathsf{maybe}(\mathit{tsig}_{\mathrm{rec}})$. Thanks to the signature peeling judgment, references to *modid* in *modexp* will be implicitly fetch'ed.

10. Perform the main phase of elaboration, producing $\mathit{mod}_{\mathrm{actual}}$ with signature $\mathit{tsig}_{\mathrm{actual}}$. Note: $\mathsf{unroll}(\mathit{mvar}_{\mathrm{static}})$.hidden tells the $\vdash_{\mathrm{rec}}$ judgment which submodule of $\mathit{mvar}_{\mathrm{static}}$ corresponds to *modexp*. (Similarly, the box in line 6 tells the $\vdash_{\mathrm{rec}}$ judgment what part of $\mathit{metasig}_{\mathrm{static}}$ corresponds to *modexp*.)

11. Match the actual signature of the body, $\mathit{tsig}_{\mathrm{actual}}$, against the required signature $\mathit{tsig}_{\mathrm{rec}}$ in order to produce a coercion module $\mathit{pmod}_{\mathrm{coerced}}$.

12. Put all the pieces together and seal the result with the original declared signature $\mathit{sig}_{\mathrm{rec}}$.

Static Phase of Recursive Module Elaboration

The output $\mathit{metaldec}(s)/\mathit{metasig}$ has the property that it is *almost* entirely static and transparent. The only construct that translates to a *metaldec* with value specifications in it is the `datatype` binding (Rule 9.158), and the only constructs that produce switchable meta-signatures are sealed module expressions and recursive module expressions (Rules 9.171 and 9.172). Otherwise, the rules are totally straightforward.

$$\boxed{\Gamma \vdash_{\mathrm{stat}} \mathit{bindings} \rightsquigarrow \mathit{metaldecs}}$$

$$\frac{}{\Gamma \vdash_{\mathrm{stat}} \cdot \rightsquigarrow \cdot} \tag{9.153}$$

$$\frac{\Gamma \vdash_{\mathrm{stat}} \mathit{sigexp} \rightsquigarrow \mathit{sig} : \mathsf{Sig} \qquad \Gamma, \overline{\mathit{sigid}}{=}\mathit{sig} \vdash_{\mathrm{stat}} \mathit{bindings} \rightsquigarrow \mathit{metaldecs}}{\Gamma \vdash_{\mathrm{stat}} \texttt{signature}\ \mathit{sigid}\ \texttt{=}\ \mathit{sigexp}\ \langle \texttt{;} \rangle\ \mathit{bindings} \rightsquigarrow \mathit{metaldecs}} \tag{9.154}$$

$$\frac{\Gamma \vdash_{\mathrm{stat}} \mathit{binding} \rightsquigarrow \mathit{metaldecs}_1 \qquad \Gamma, \mathit{metaldecs}_1 \vdash_{\mathrm{stat}} \mathit{bindings} \rightsquigarrow \mathit{metaldecs}_2}{\Gamma \vdash_{\mathrm{stat}} \mathit{binding}\ \langle \texttt{;} \rangle\ \mathit{bindings} \rightsquigarrow \mathit{metaldecs}_1 {+}{+} \mathit{metaldecs}_2} \tag{9.155}$$

$$\boxed{\Gamma \vdash_{\mathrm{stat}} \mathit{binding} \rightsquigarrow \mathit{metaldecs}}$$

$$\frac{\mathit{binding}\ \text{is a}\ \texttt{val}\ \text{or}\ \texttt{exception}\ \text{binding.}}{\Gamma \vdash_{\mathrm{stat}} \mathit{binding} \rightsquigarrow \cdot} \tag{9.156}$$

$$\frac{\mathit{binding}\ \text{is a}\ \texttt{type}\ \text{or}\ \texttt{open}\ \text{binding.} \qquad \Gamma \vdash \mathit{binding} \rightsquigarrow \_ :_{\mathrm{P}} \mathit{tldecs}}{\Gamma \vdash_{\mathrm{stat}} \mathit{binding} \rightsquigarrow \mathsf{Stat}(\mathit{tldecs})} \tag{9.157}$$

$$\frac{\Gamma \vdash \mathit{datbinds} \rightsquigarrow \mathit{sig}}{\Gamma \vdash_{\mathrm{stat}} \texttt{datatype}\ \mathit{datbinds} \rightsquigarrow \mathit{ilab}^{*}{:}\mathit{sig}} \tag{9.158}$$

Rule 9.158: It is important here that the signature be *sig*, not $\mathsf{Stat}(\mathit{sig})$, since *sig* has a canonical implementation while $\mathsf{Stat}(\mathit{sig})$ does not.

$$\frac{\Gamma \vdash_{\overline{\mathrm{ctx}}} \overline{longconid} \rightsquigarrow \_ : tknd}{\Gamma \vdash_{\overline{\mathrm{stat}}} \texttt{datatype}\ conid\ \texttt{=}\ \texttt{datatype}\ longconid \rightsquigarrow \overline{conid}{:}tknd} \tag{9.159}$$

$$\frac{\Gamma \vdash_{\overline{\mathrm{stat}}} bindings_1 \rightsquigarrow metaldecs_1 \qquad \Gamma, ilab^{*}{\triangleright}mvar{:}[\![metaldecs_1]\!] \vdash_{\overline{\mathrm{stat}}} bindings_2 \rightsquigarrow metaldecs_2}{\Gamma \vdash_{\overline{\mathrm{stat}}} \texttt{local}\ bindings_1\ \texttt{in}\ bindings_2\ \texttt{end} \rightsquigarrow ilab{\triangleright}mvar{:}[\![metaldecs_1]\!], metaldecs_2} \tag{9.160}$$

$$\frac{\Gamma \vdash_{\overline{\mathrm{stat}}} modexp \rightsquigarrow metasig}{\Gamma \vdash_{\overline{\mathrm{stat}}} \texttt{module}\ modid\ \texttt{=}\ modexp \rightsquigarrow \overline{modid}{:}metasig} \tag{9.161}$$

$\boxed{\Gamma \vdash_{\overline{\mathrm{stat}}} modexp \rightsquigarrow metasig}$

$$\frac{\Gamma \vdash_{\overline{\mathrm{ctx}}} \overline{longmodid} \rightsquigarrow pmod : tsig}{\Gamma \vdash_{\overline{\mathrm{stat}}} longmodid \rightsquigarrow \mathsf{Stat}(tsig)} \tag{9.162}$$

$$\frac{\Gamma \vdash_{\overline{\mathrm{stat}}} bindings \rightsquigarrow metaldecs}{\Gamma \vdash_{\overline{\mathrm{stat}}} \texttt{struct}\ bindings\ \texttt{end} \rightsquigarrow [\![metaldecs]\!]} \tag{9.163}$$

$$\frac{\Gamma \vdash_{\overline{\mathrm{stat}}} modexp \rightsquigarrow metasig \qquad \Gamma, mvar{:}metasig; mvar{:}metasig \vdash_{\overline{\mathrm{sig}}} \overline{modid} \rightsquigarrow pmod : tsig \qquad modexp \text{ is not of the form } longmodid}{\Gamma \vdash_{\overline{\mathrm{stat}}} modexp\,\texttt{.}\,modid \rightsquigarrow \exists(mvar{:}metasig).\mathsf{Stat}(tsig)} \tag{9.164}$$

$$\frac{\Gamma \vdash_{\overline{\mathrm{stat}}} sigexp \rightsquigarrow statsig : \mathsf{Sig} \qquad \Gamma, \overline{modid}{\triangleright}mvar{:}statsig \vdash_{\overline{\mathrm{stat}}} modexp \rightsquigarrow metasig}{\Gamma \vdash_{\overline{\mathrm{stat}}} \texttt{functor (}modid\ \texttt{:}\ sigexp\texttt{) ->}\ modexp \rightsquigarrow \Pi^{\mathsf{tot}}(mvar{:}statsig).metasig} \tag{9.165}$$

$$\frac{}{\Gamma \vdash_{\overline{\mathrm{stat}}} \texttt{functor (}modid\ \texttt{:}\ sigexp\texttt{) ->>}\ modexp \rightsquigarrow [\![\cdot]\!]} \tag{9.166}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\overline{\mathrm{stat}}} modexp_1 \rightsquigarrow metasig_1 \qquad \Gamma \vdash_{\overline{\mathrm{stat}}} modexp_2 \rightsquigarrow metasig_2 \\ \Gamma, mvar_1{:}metasig_1 \vdash_{\overline{\mathrm{peel}}} mvar_1{:}metasig_1 \rightsquigarrow \_ : \Pi^{\mathsf{tot}}(mvar{:}statsig').tsig'' \\ \Gamma, mvar_1{:}metasig_1, mvar_2{:}metasig_2 \vdash_{\overline{\mathrm{sub}}} mvar_2 : metasig_2 \preceq statsig' \rightsquigarrow pmod : \_\end{array}}{\begin{array}{c}\Gamma \vdash_{\overline{\mathrm{stat}}} modexp_1\texttt{(}modexp_2\texttt{)} \rightsquigarrow \exists(mvar_1{:}metasig_1).\exists(mvar_2{:}metasig_2). \\ \mathsf{Stat}(tsig''[pmod/mvar])\end{array}} \tag{9.167}$$

$$\frac{\Gamma \vdash_{\overline{\mathrm{stat}}} bindings \rightsquigarrow metaldecs \qquad \Gamma, ilab^{*}{\triangleright}mvar{:}[\![metaldecs]\!] \vdash_{\overline{\mathrm{stat}}} modexp \rightsquigarrow metasig}{\Gamma \vdash_{\overline{\mathrm{stat}}} \texttt{let}\ bindings\ \texttt{in}\ modexp\ \texttt{end} \rightsquigarrow \exists(mvar{:}[\![metaldecs]\!]).metasig} \tag{9.168}$$

$$\frac{\Gamma \vdash_{\overline{\mathrm{stat}}} sigexp \rightsquigarrow tstatsig : \mathsf{Sig}}{\Gamma \vdash_{\overline{\mathrm{stat}}} modexp\ seal\ sigexp \rightsquigarrow tstatsig} \tag{9.169}$$

$$\frac{\begin{array}{c}\text{Rule 9.169 does not apply.}\\ \Gamma \vdash_{\text{stat}} \mathit{modexp} \rightsquigarrow \mathit{metasig} \qquad \Gamma \vdash_{\text{stat}} \mathit{sigexp} \rightsquigarrow \mathit{statsig} : \mathsf{Sig}\\ \Gamma, \mathit{mvar}{:}\mathit{metasig} \vdash_{\text{sub}} \mathit{mvar} : \mathit{metasig} \preceq \mathit{statsig} \rightsquigarrow \_ : \mathit{tsig}\end{array}}{\Gamma \vdash_{\text{stat}} \mathit{modexp}\ \texttt{:}\ \mathit{sigexp} \rightsquigarrow \exists(\mathit{mvar}{:}\mathit{metasig}).\mathit{tsig}} \tag{9.170}$$

$$\frac{\begin{array}{c}\text{Rule 9.169 does not apply.}\\ \Gamma \vdash_{\text{stat}} \mathit{modexp} \rightsquigarrow \mathit{metasig} \qquad \Gamma \vdash_{\text{stat}} \mathit{sigexp} \rightsquigarrow \mathit{statsig} : \mathsf{Sig}\\ \Gamma, \mathit{mvar}{:}\mathit{metasig} \vdash_{\text{sub}} \mathit{mvar} : \mathit{metasig} \preceq \mathit{statsig} \rightsquigarrow \_ : \mathit{tsig}\end{array}}{\Gamma \vdash_{\text{stat}} \mathit{modexp}\ \texttt{:>}\ \mathit{sigexp} \rightsquigarrow \exists(\mathit{mvar}{:}\mathit{metasig}).\{\text{private}{=}\mathit{tsig}, \text{public}{=}\mathit{statsig}\}} \tag{9.171}$$

Rule 9.171: Note the similarity between this rule and the previous one. The only difference is that the switchable signature output by this rule offers a public, opaque view of *modexp* in addition to the private, transparent one. Also note that there is no rule for impure sealing (unless the sealing signature is transparent), since the body of a recursive module is required to be pure/separable.

$$\frac{\begin{array}{l}\Gamma \vdash_{\text{stat}} \mathit{sigexp} \rightsquigarrow \mathit{statsig}_{\text{rec}} : \mathsf{Sig}\\ \Gamma, \overline{\mathit{modid}}{\triangleright}\mathit{mvar}_{\text{rec}}{:}\mathit{statsig}_{\text{rec}} \vdash_{\text{stat}} \mathit{modexp} \rightsquigarrow \mathit{metasig}_{\text{actual}}\\ \mathit{mvar}^{\mathsf{c}}_{\text{rec}} \notin \mathrm{FV}(\mathsf{Stat}(\mathsf{Priv}(\mathit{metasig}_{\text{actual}}))) \cup \mathrm{FV}(\mathsf{Stat}(\mathsf{Pub}(\mathit{metasig}_{\text{actual}})))\\ \Gamma, \mathit{mvar}_{\text{actual}}{:}\mathsf{Stat}(\mathsf{Pub}(\mathit{metasig}_{\text{actual}})) \vdash_{\text{sub}}\\ \qquad \mathit{mvar}_{\text{actual}} : \mathsf{Stat}(\mathsf{Pub}(\mathit{metasig}_{\text{actual}})) \preceq \mathit{statsig}_{\text{rec}} \rightsquigarrow \_ : \mathit{tstatsig}_{\text{coerced}}\\ \mathit{metasig}'_{\text{actual}} := \mathit{metasig}_{\text{actual}}[\mathit{mvar}^{\mathsf{c}}_{\text{rec}}.\text{visible}^{*}/\mathit{mvar}^{\mathsf{c}}_{\text{rec}}]\\ \mathit{metasig}_{\text{static}} := \rho(\mathit{mvar}_{\text{rec}}).\exists(\mathit{mvar}_{\text{actual}}{:}\mathit{metasig}'_{\text{actual}}).\\ \qquad\qquad\qquad\qquad\qquad\qquad \{\text{private}{=}\mathit{tstatsig}_{\text{coerced}}, \text{public}{=}\mathit{statsig}_{\text{rec}}\}\end{array}}{\Gamma \vdash_{\text{stat}} \texttt{rec (}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \mathit{modexp} \rightsquigarrow \mathit{metasig}_{\text{static}}} \tag{9.172}$$

Rule 9.172: The premises of this rule are almost identical to the first six premises of Rule 9.152, which makes sense since static elaboration is the first phase of recursive module elaboration. The only significant difference is that here $\mathit{metasig}_{\text{static}}$ offers a public, opaque view of the recursive module in addition to the private, transparent one.

Main Phase of Recursive Module Elaboration

The judgments describing the main phase of recursive module elaboration make use of a special "meta-context" Θ of the form $\Gamma; \mathit{metadec}; \Gamma'$. Here, *metadec* is a declaration of the form $\mathit{mvar}_{\text{static}}{:}\mathit{metasig}_{\text{static}}$, where $\mathit{metasig}_{\text{static}}$ represents the product of the static phase of recursive module elaboration. It is important that Θ contain the full $\mathit{metasig}_{\text{static}}$ and not just its public setting, because we will want to actually switch some of the settings in $\mathit{metasig}_{\text{static}}$ from public to private during elaboration. Wherever I use a meta-context $\Theta = \Gamma; \mathit{metadec}; \Gamma'$ in a premise that expects a normal elaboration context, Θ should be implicitly erased to $\Gamma, \mathsf{Pub}(\mathit{metadec}), \Gamma'$.

What makes the main phase somewhat tricky to formalize is that we need to keep track of what part of $\mathit{metasig}_{\text{static}}$ (and, similarly, what projection from $\mathit{mvar}_{\text{static}}$) corresponds to the piece of the recursive module body we are currently elaborating. The translation judgments have the form $\Theta \vdash_{\text{rec}} \mathit{pmod} \Rightarrow \mathit{binding(s)} \rightsquigarrow \mathit{lbnds} : \mathit{tldecs}$ and $\Theta \vdash_{\text{rec}} \mathit{pmod} \Rightarrow \mathit{modexp} \rightsquigarrow \mathit{mod} : \mathit{tsig}$. The input *pmod* tells us where we are in $\mathit{mvar}_{\text{static}}$. Specifically, *pmod* will have the form of a "path"—*i.e.,* a sequence of eliminations (*i.e.,* projections, applications, unroll'ings)—headed by $\mathit{mvar}_{\text{static}}$.

To indicate where we are in $metasig_{\text{static}}$, I will employ the simple, if unusual, technique of surrounding it with a box (literally). When the judgment makes a recursive call on a subterm, the box will shrink in order to enclose only the meta-signature/meta-declaration(s) in $metasig_{\text{static}}$ corresponding to that subterm. In many of the rules, it is useful to be able to "zoom in" on the part of $metasig_{\text{static}}$ that is boxed. To enable this, I will write $metadec\{\boxed{metasig}\}$ (resp. $metadec\{\boxed{metaldecs}\}$) to signify that $metasig$ (resp. $metaldecs$) is the boxed part of $metasig_{\text{static}}$.

Aside from keeping track of where we are in $metasig_{\text{static}}$, which is totally straightforward, most of the rules are very similar to the normal elaboration rules for bindings and module expressions. The constructs whose rules are most interesting are those that involve some form of data abstraction—namely, `datatype` bindings, sealed module expressions and recursive module expressions (Rules 9.178, 9.188 and 9.189). While recursive module elaboration respects data abstraction boundaries during typechecking, the *output* of recursive module elaboration is transparent, so that the recursive module body will match the transparent declared signature ($tsig_{\text{rec}}$ in Rule 9.152).

$$\boxed{\Theta \vdash_{\text{rec}} pmod \Rightarrow bindings \rightsquigarrow lbnds : tldecs}$$

$$\frac{\Theta = \Gamma; metadec\{\boxed{\cdot}\}; \Gamma'}{\Theta \vdash_{\text{rec}} pmod \Rightarrow \cdot \rightsquigarrow \cdot : \cdot} \tag{9.173}$$

$$\frac{\Theta \vdash sigexp \rightsquigarrow sig : \mathsf{Sig} \qquad \Theta, \overline{sigid}{=}sig \vdash_{\text{rec}} pmod \Rightarrow bindings \rightsquigarrow lbnds : tldecs}{\Theta \vdash_{\text{rec}} pmod \Rightarrow \texttt{signature}\ sigid\ \texttt{=}\ sigexp\ \langle\texttt{;}\rangle\ bindings \rightsquigarrow lbnds : tldecs} \tag{9.174}$$

$$\frac{\begin{array}{c}\Theta = \Gamma; metadec\{\boxed{metaldecs_1, metaldecs_2}\}; \Gamma' \\ \Gamma; metadec\{\boxed{metaldecs_1}, metaldecs_2\}; \Gamma' \vdash_{\text{rec}} pmod \Rightarrow binding \rightsquigarrow lbnds_1 : tldecs_1 \\ \Gamma; metadec\{metaldecs_1, \boxed{metaldecs_2}\}; \Gamma', tldecs_1 \vdash_{\text{rec}} pmod \Rightarrow bindings \rightsquigarrow lbnds_2 : tldecs_2\end{array}}{\Theta \vdash_{\text{rec}} pmod \Rightarrow binding\ \langle\texttt{;}\rangle\ bindings \rightsquigarrow lbnds_1{+\!\!+}lbnds_2 : tldecs_1{+\!\!+}tldecs_2} \tag{9.175}$$

$$\boxed{\Theta \vdash_{\text{rec}} pmod \Rightarrow binding \rightsquigarrow lbnds : tldecs}$$

$$\frac{\begin{array}{c}binding\ \text{is a}\ \texttt{val}\ \text{or}\ \texttt{exception}\ \text{binding.} \\ \Theta = \Gamma; metadec\{\boxed{\cdot}\}; \Gamma' \\ \Theta \vdash binding \rightsquigarrow lbnds :_{\mathtt{P}} tldecs\end{array}}{\Theta \vdash_{\text{rec}} pmod \Rightarrow binding \rightsquigarrow lbnds : tldecs} \tag{9.176}$$

Rule 9.176: For `val` and `exception` bindings, there is no static part (the box in Θ is empty), and we default to using normal elaboration.

$$\frac{\begin{array}{c}binding\ \text{is a}\ \texttt{type},\ \texttt{open}\ \text{or}\ \texttt{datatype}\ \text{replication binding.} \\ \Theta = \Gamma; metadec\{\boxed{metaldecs}\}; \Gamma' \\ \Theta \vdash binding \rightsquigarrow lbnds :_{\mathtt{P}} tldecs \\ \text{length}(metaldecs) = \text{length}(tldecs)\end{array}}{\Theta \vdash_{\text{rec}} pmod \Rightarrow binding \rightsquigarrow lbnds : tldecs} \tag{9.177}$$

Rule 9.177: For these atomic, transparent bindings, we can default to normal elaboration. The last premise ensures that the box in Θ encloses the right number of *metaldecs*.

$$\frac{\begin{array}{c}\Theta = \Gamma; \mathit{metadec}\{\boxed{\mathit{ilab}^*{:}\mathit{sig}'}\}; \Gamma' \\ \Theta \vdash \mathit{datbinds} \rightsquigarrow \mathit{sig} \qquad \Theta \vdash \mathit{pmod}.\mathit{ilab}^* : \mathit{sig}\end{array}}{\begin{array}{l}\Theta \vdash_{\mathrm{rec}} \mathit{pmod} \Rightarrow \texttt{datatype}\ \mathit{datbinds} \rightsquigarrow \\ \qquad \mathit{ilab}^*{=}\mathit{pmod}.\mathit{ilab}^* : \mathit{ilab}^*{:}\mathfrak{S}(\mathit{pmod}.\mathit{ilab}^* : \mathit{sig})\end{array}} \tag{9.178}$$

Rule 9.178: For a `datatype` binding, we know that the static phase has already computed the `datatype` module. We can therefore copy the module directly from *pmod* (cf. Figure 5.20).

$$\frac{\begin{array}{l}\qquad\qquad \Theta = \Gamma; \mathit{metadec}\{\boxed{\mathit{ilab}{:}[\![\mathit{metaldecs}_1]\!], \mathit{metaldecs}_2}\}; \Gamma' \\ \Gamma; \mathit{metadec}\{\mathit{ilab}{:}[\![\boxed{\mathit{metaldecs}_1}]\!], \mathit{metaldecs}_2\}; \Gamma' \vdash_{\mathrm{rec}} \\ \quad \mathit{pmod}.\mathit{ilab} \Rightarrow \mathit{bindings}_1 \rightsquigarrow \mathit{lbnds}_1 : \mathit{tldecs}_1 \\ \Gamma; \mathit{metadec}\{\mathit{ilab}{:}[\![\mathit{metaldecs}_1]\!], \boxed{\mathit{metaldecs}_2}\}; \Gamma', \mathit{ilab}^*{\triangleright}\mathit{mvar}{:}[\![\mathit{tldecs}_1]\!] \vdash_{\mathrm{rec}} \\ \quad \mathit{pmod} \Rightarrow \mathit{bindings}_2 \rightsquigarrow \mathit{lbnds}_2 : \mathit{tldecs}_2\end{array}}{\begin{array}{l}\Theta \vdash_{\mathrm{rec}} \mathit{pmod} \Rightarrow \texttt{local}\ \mathit{bindings}_1\ \texttt{in}\ \mathit{bindings}_2\ \texttt{end} \rightsquigarrow \\ \qquad \mathit{ilab}{\triangleright}\mathit{mvar}{=}[\mathit{lbnds}_1], \mathit{lbnds}_2 : \mathit{ilab}{\triangleright}\mathit{mvar}{:}[\![\mathit{tldecs}_1]\!], \mathit{tldecs}_2\end{array}} \tag{9.179}$$

$$\frac{\begin{array}{c}\Theta = \Gamma; \mathit{metadec}\{\boxed{\mathit{lab}{:}\mathit{metasig}}\}; \Gamma' \\ \Gamma; \mathit{metadec}\{\mathit{lab}{:}\boxed{\mathit{metasig}}\}; \Gamma' \vdash_{\mathrm{rec}} \mathit{pmod}.\mathit{lab} \Rightarrow \mathit{modexp} \rightsquigarrow \mathit{mod} : \mathit{tsig}\end{array}}{\Theta \vdash_{\mathrm{rec}} \mathit{pmod} \Rightarrow \texttt{module}\ \mathit{modid}\ \texttt{=}\ \mathit{modexp} \rightsquigarrow \overline{\mathit{modid}}{=}\mathit{mod} : \overline{\mathit{modid}}{:}\mathit{tsig}} \tag{9.180}$$

$$\boxed{\Theta \vdash_{\mathrm{rec}} \mathit{pmod} \Rightarrow \mathit{modexp} \rightsquigarrow \mathit{mod} : \mathit{tsig}}$$

$$\frac{\begin{array}{c}\Theta = \Gamma; \mathit{metadec}\{\boxed{\mathit{tsig}'}\}; \Gamma' \\ \Theta \vdash \mathit{modexp} \rightsquigarrow \mathit{mod} :_{\kappa} \mathit{tsig}\end{array}}{\Theta \vdash_{\mathrm{rec}} \mathit{pmod} \Rightarrow \mathit{modexp} \rightsquigarrow \mathsf{purify}(\mathit{mod}) : \mathit{tsig}} \tag{9.181}$$

Rule 9.181: If the boxed signature in Θ is a normal transparent signature, then we can use the same Θ for typechecking all of *modexp* and default to normal elaboration.

Rules 9.182–9.188 assume that the highlighted signature in Θ is not transparent, in which case Rule 9.181 does not apply.

$$\frac{\begin{array}{c}\Theta = \Gamma; \mathit{metadec}\{\boxed{[\![\mathit{metaldecs}]\!]}\}; \Gamma' \\ \Gamma; \mathit{metadec}\{[\![\boxed{\mathit{metaldecs}}]\!]\}; \Gamma' \vdash_{\mathrm{rec}} \mathit{pmod} \Rightarrow \mathit{bindings} \rightsquigarrow \mathit{lbnds} : \mathit{tldecs}\end{array}}{\Theta \vdash_{\mathrm{rec}} \mathit{pmod} \Rightarrow \texttt{struct}\ \mathit{strdec}\ \texttt{end} \rightsquigarrow [\mathit{lbnds}] : [\![\mathit{tldecs}]\!]} \tag{9.182}$$

$$\frac{\begin{array}{c}\Theta = \Gamma; \mathit{metadec}\{\boxed{\exists(\mathit{mvar}{:}\mathit{metasig}).\mathit{sig}}\}; \Gamma' \\ \Gamma; \mathit{metadec}\{\exists(\mathit{mvar}{:}\boxed{\mathit{metasig}}).\mathit{sig}\}; \Gamma' \vdash_{\mathrm{rec}} \mathit{pmod}.\mathrm{hidden} \Rightarrow \mathit{modexp} \rightsquigarrow \mathit{mod} : \mathit{tsig} \\ \Theta, \mathit{mvar}{:}\mathit{tsig}; \mathit{mvar}{:}\mathit{tsig} \vdash_{\overline{\mathrm{sig}}} \overline{\mathit{modid}} \rightsquigarrow \mathit{pmod}' : \mathit{tsig}'\end{array}}{\Theta \vdash_{\mathrm{rec}} \mathit{pmod} \Rightarrow \mathit{modexp}\,\texttt{.}\,\mathit{modid} \rightsquigarrow \mathsf{elet}\ \mathit{mvar}{=}\mathit{mod}\ \mathsf{in}\ \mathit{pmod}' : \exists(\mathit{mvar}{:}\mathit{tsig}).\mathit{tsig}'} \tag{9.183}$$

$$\frac{\begin{array}{c}\Theta = \Gamma; metadec\{\boxed{\Pi^{\mathsf{tot}}(mvar{:}sig').metasig}\}; \Gamma' \\ \Theta \vdash sigexp \rightsquigarrow sig : \mathsf{Sig} \\ \Gamma; metadec\{\Pi^{\mathsf{tot}}(mvar{:}sig').\boxed{metasig}\}; \Gamma', \overline{modid}{\triangleright}mvar{:}sig \vdash_{\mathrm{rec}} \\ \quad pmod(mvar) \Rightarrow modexp \rightsquigarrow mod : tsig\end{array}}{\begin{array}{c}\Theta \vdash_{\mathrm{rec}} pmod \Rightarrow \texttt{functor (}modid\texttt{ : }sigexp\texttt{) -> }modexp \rightsquigarrow \\ \lambda^{\mathsf{tot}}(mvar{:}sig).mod : \Pi^{\mathsf{tot}}(mvar{:}sig).tsig\end{array}} \tag{9.184}$$

Rule 9.184: This rule handles total functors. Partial functors are handled by Rule 9.181, since a partial functor is considered transparent.

$$\frac{\begin{array}{c}\Theta = \Gamma; metadec\{\boxed{\exists(mvar_1{:}metasig_1).\exists(mvar_2{:}metasig_2).sig}\}; \Gamma' \\ \Gamma; metadec\{\exists(mvar_1{:}\boxed{metasig_1}).\exists(mvar_2{:}metasig_2).sig\}; \Gamma' \vdash_{\mathrm{rec}} \\ \quad pmod.\mathrm{hidden} \Rightarrow modexp_1 \rightsquigarrow mod_1 : tsig_1 \\ \Gamma; metadec\{\exists(mvar_1{:}metasig_1).\exists(mvar_2{:}\boxed{metasig_2}).sig\}; \Gamma' \vdash_{\mathrm{rec}} \\ \quad pmod.\mathrm{visible}^*.\mathrm{hidden} \Rightarrow modexp_2 \rightsquigarrow mod_2 : tsig_2 \\ \Theta, mvar_1{:}tsig_1 \vdash_{\mathrm{peel}} mvar_1{:}tsig_1 \rightsquigarrow pmod_1 : \Pi^{\mathsf{tot}}(mvar{:}sig').tsig'' \\ \Theta, mvar_1{:}tsig_1, mvar_2{:}tsig_2 \vdash_{\mathrm{sub}} mvar_2 : tsig_2 \preceq sig' \rightsquigarrow pmod_2 : \_\end{array}}{\begin{array}{l}\Theta \vdash_{\mathrm{rec}} pmod \Rightarrow modexp_1\texttt{(}modexp_2\texttt{)} \rightsquigarrow \\ \quad \mathsf{elet}\ mvar_1{=}mod_1\ \mathsf{in}\ \mathsf{elet}\ mvar_2{=}mod_2\ \mathsf{in}\ {pmod_1}^{\mathsf{tot}}(pmod_2) : \\ \quad \exists(mvar_1{:}tsig_1).\exists(mvar_2{:}tsig_2).tsig''[pmod_2/mvar]\end{array}} \tag{9.185}$$

Rule 9.185: We only allow applications of total functors; partial functor applications are impure.

$$\frac{\begin{array}{c}\Theta = \Gamma; metadec\{\boxed{\exists(mvar{:}[\![metaldecs]\!]).metasig}\}; \Gamma' \\ \Gamma; metadec\{\exists(mvar{:}[\![\boxed{metaldecs}]\!]).metasig\}; \Gamma' \vdash_{\mathrm{rec}} \\ \quad pmod.\mathrm{hidden} \Rightarrow bindings \rightsquigarrow lbnds : tldecs \\ \Gamma; metadec\{\exists(mvar{:}[\![metaldecs]\!]).\boxed{metasig}\}; \Gamma', ilab^*{\triangleright}mvar{:}[\![tldecs]\!] \vdash_{\mathrm{rec}} \\ \quad pmod.\mathrm{visible}^* \Rightarrow modexp \rightsquigarrow mod : tsig\end{array}}{\begin{array}{c}\Theta \vdash_{\mathrm{rec}} pmod \Rightarrow \texttt{let }bindings\texttt{ in }modexp\texttt{ end} \rightsquigarrow \\ \mathsf{elet}\ mvar{=}[lbnds]\ \mathsf{in}\ mod : \exists(mvar{:}[\![tldecs]\!]).tsig\end{array}} \tag{9.186}$$

$$\frac{\begin{array}{c}\Theta = \Gamma; metadec\{\boxed{\exists(mvar{:}metasig).sig'}\}; \Gamma' \\ \Gamma; metadec\{\exists(mvar{:}\boxed{metasig}).sig'\}; \Gamma' \vdash_{\mathrm{rec}} pmod.\mathrm{hidden} \Rightarrow modexp \rightsquigarrow mod : tsig \\ \Theta \vdash sigexp \rightsquigarrow sig : \mathsf{Sig} \\ \Theta, mvar{:}tsig \vdash_{\mathrm{sub}} mvar : tsig \preceq sig \rightsquigarrow pmod' : tsig'\end{array}}{\Theta \vdash_{\mathrm{rec}} pmod \Rightarrow modexp\ \texttt{:}\ sigexp \rightsquigarrow \mathsf{elet}\ mvar{=}mod\ \mathsf{in}\ pmod' : \exists(mvar{:}tsig).tsig'} \tag{9.187}$$

$$\frac{\begin{array}{c}\Theta = \Gamma; \mathit{metadec}\{\boxed{\exists(\mathit{mvar}{:}\mathit{metasig}).\{\text{private}{=}\mathit{tsig}'_{\text{private}}, \text{public}{=}\mathit{sig}'_{\text{public}}\}}\}; \Gamma' \\ \Theta \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig}_{\text{public}} : \mathsf{Sig} \\ \mathit{tsig}_{\text{public}} := \mathfrak{S}(\mathit{pmod}.\text{visible}^* : \mathit{sig}_{\text{public}}) \qquad \Theta \vdash \mathsf{Fst}(\mathit{pmod}).\text{visible}^* : \mathsf{Fst}(\mathit{sig}_{\text{public}}) \\ \Theta_{\text{private}} := \Gamma; \mathit{metadec}\{\exists(\mathit{mvar}{:}\boxed{\mathit{metasig}}).\mathit{tsig}'_{\text{private}}\}; \Gamma' \\ \Theta_{\text{private}} \vdash_{\text{rec}} \mathit{pmod}.\text{hidden} \Rightarrow \mathit{modexp} \rightsquigarrow \mathit{mod}_{\text{actual}} : \mathit{tsig}_{\text{actual}} \\ \Theta_{\text{private}}, \mathit{mvar}{:}\mathit{tsig}_{\text{actual}} \vdash_{\text{sub}} \mathit{mvar} : \mathit{tsig}_{\text{actual}} \preceq \mathit{tsig}_{\text{public}} \rightsquigarrow \mathit{mod}_{\text{coerced}} : \_\end{array}}{\begin{array}{l}\Theta \vdash_{\text{rec}} \mathit{pmod} \Rightarrow \mathit{modexp}\ \texttt{:>}\ \mathit{sigexp} \rightsquigarrow \\ \qquad \mathsf{let}\ \mathit{mvar}{=}\mathit{mod}_{\text{actual}}\ \mathsf{in}\ (\mathit{mod}_{\text{coerced}} : \mathit{tsig}_{\text{public}}) : \mathit{tsig}_{\text{public}}\end{array}} \tag{9.188}$$

Rule 9.188: Two important points: (1) When we go beneath the sealing to elaborate *modexp*, we use a modified meta-context Θ_{private} in which the public signature of *pmod*.visible* is eliminated and only the private one remains. Θ_{private} exposes the implementation of *modexp* by allowing us to observe the connection between the type components of *pmod*.visible* and *pmod*.hidden that Θ obscured. (2) The output signature is not $\mathit{sig}_{\text{public}}$ but rather the selfification of $\mathit{sig}_{\text{public}}$ with respect to *pmod*.visible* (cf. Figure 5.22). The purpose of this is to allow the code outside of *modexp* `:>` *sigexp* to observe that the type components of this module expression are equivalent to those of *pmod*.visible*, thus avoiding the double vision problem. However, the private implementation of *modexp* is still kept hidden, because the signature that the rest of the recursive module body sees for *pmod*.visible* is the public one.

$$\frac{\begin{array}{c}\Theta = \Gamma; \mathit{metadec}\{\boxed{\rho(\mathit{mvar}_{\text{rec}}).\exists(\mathit{mvar}{:}\mathit{metasig}).\{\text{private}{=}\mathit{tsig}'_{\text{private}}, \text{public}{=}\mathit{sig}'_{\text{public}}\}}\}; \Gamma' \\ \Theta \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig}_{\text{rec}} : \mathsf{Sig} \\ \mathit{tsig}_{\text{rec}} := \mathfrak{S}(\mathsf{unroll}(\mathit{pmod}).\text{visible}^* : \mathit{sig}_{\text{rec}}) \qquad \Theta \vdash \mathsf{Fst}(\mathit{pmod}).\text{visible}^* : \mathsf{Fst}(\mathit{sig}_{\text{rec}}) \\ \Theta_{\text{private}} := \Gamma; \mathit{metadec}\{\rho(\mathit{mvar}_{\text{rec}}).\exists(\mathit{mvar}{:}\boxed{\mathit{metasig}}).\mathit{tsig}'_{\text{private}}\} \\ ; \Gamma', \overline{\mathit{modid}} \triangleright \mathit{mvar}_{\text{rec}}{:}\mathsf{maybe}(\mathit{tsig}_{\text{rec}}) \\ \Theta_{\text{private}} \vdash_{\text{rec}} \mathsf{unroll}(\mathit{pmod}).\text{hidden} \Rightarrow \mathit{modexp} \rightsquigarrow \mathit{mod}_{\text{actual}} : \mathit{tsig}_{\text{actual}} \\ \Theta_{\text{private}}, \mathit{mvar}_{\text{actual}}{:}\mathit{tsig}_{\text{actual}} \vdash_{\text{sub}} \mathit{mvar}_{\text{actual}} : \mathit{tsig}_{\text{actual}} \preceq \mathit{tsig}_{\text{rec}} \rightsquigarrow \mathit{pmod}_{\text{coerced}} : \_\end{array}}{\begin{array}{l}\Theta \vdash_{\text{rec}} \mathit{pmod} \Rightarrow \texttt{rec (}\mathit{modid}\ \texttt{:}\ \mathit{sigexp}\texttt{)}\ \mathit{modexp} \rightsquigarrow \\ \qquad \mathsf{rec}(\mathit{mvar}_{\text{rec}}{:}\mathit{tsig}_{\text{rec}}.\mathsf{let}\ \mathit{mvar}_{\text{actual}}{=}\mathit{mod}_{\text{actual}}\ \mathsf{in}\ (\mathit{pmod}_{\text{coerced}} : \mathit{tsig}_{\text{rec}})) : \mathit{tsig}_{\text{rec}}\end{array}} \tag{9.189}$$

Rule 9.189: Just as the static elaboration rule for recursive modules performed the first half of Rule 9.152, the present rule performs the last half. The only additional point is that recursive module expressions are implicitly sealed with their declared signatures. Thus, when elaboration goes inside the recursive module, we must modify the context (in the same way as in Rule 9.188) in order to expose the private connection between the type components of $\mathsf{unroll}(\mathit{pmod})$.visible* and $\mathsf{unroll}(\mathit{pmod})$.hidden.

9.4 Notes on Implementation

A clear, formal language definition is indisputably useful, but it can also conceal certain issues that cause serious problems for practical implementation. In order to demonstrate the viability of my language design ideas, I have implemented, as an experimental alternate front end to the TILT compiler, a language similar to the one defined in this chapter. The language I implemented is not exactly the same as the one presented here, partly because the implementation was based on an earlier formalization and partly due to time constraints on my implementation work. Fortunately, the recursive module and recursively dependent signature constructs, which are arguably the most useful and semantically complex of all the new features in my language, are implemented in a way that follows the present formalization very closely. Furthermore, my limited testing has shown the implementation to exhibit the intended behavior on all the examples of Chapter 5.

Most of the differences between my implementation and the present formalization are with regard to superficial choices of surface syntax. In particular, while I have felt free in this chapter to revise SML syntax as I see fit—*e.g.,* supporting `module` bindings as opposed to SML's `structure` and `functor` bindings—my implementation sticks closer to SML syntax in order to provide backward compatibility with existing SML code bases. However, there are a few more significant distinctions worth mentioning.

First, my implementation does not support the `packtype` mechanism, nor does it provide any other mechanism for packaging modules as first-class values. The reason for this is pragmatic: whereas I found that recursive modules constituted a relatively orthogonal extension to the TILT compiler (and mostly just to the front end), supporting package types and first-class module values would have required me to make major changes to the whole compiler.

To understand why, observe that in the absence of `pack` *mod* `as` *sig* and `unpack` *exp* `as` *sig*, the IL has a purely second-class module system. That is to say, the type system may distinguish "separable" modules from "inseparable" modules, but in truth there is no way for the underlying type components of a module to depend on any dynamic values or effects. As a consequence, it is possible to phase-split *all* modules, not just separable modules, into a type constructor and a term, by simply ignoring all uses of sealing and translating partial functors in the same way as total functors. As Standard ML has a purely second-class module language, the original TILT front end takes precisely this approach, and all the intermediate stages of the compiler are designed to work with code that has been phase-split in this way. Rather than change an assumption of the whole compiler, I opted to follow TILT's approach to phase-splitting and omit first-class module packages from the implementation, at least for the time being.

Second, as I discussed in Section 2.2.1, an unfortunate deficiency of all existing dialects of the ML module system (including the one in this thesis) is their lack of support for inseparable sealing and statically total functors. In the earlier language design on which my implementation is based, I attempted to remedy this deficiency. My idea was to have separably total functors behave more like statically total functors by making static module equivalence so conservative that it implies dynamic equivalence. Specifically, I had the elaborator automatically insert into every structure expression, `struct` *bindings* `end`, a sealed submodule defining an abstract "identity" type. The effect of these identity types is to render two modules statically equivalent only if their identity types are equivalent, which in turn is only the case if they are syntactically identical or one is a renaming of the other (*e.g.,* `structure X = Y`). In essence, this is a generalization of the approach taken by O'Caml, which is not as brittle as O'Caml's syntactic equivalence because it allows for renamings of modules.

However, as I also discussed in Section 2.2.1, syntactic equivalence does not always imply dynamic equivalence. For instance, the functor application `F(X)` is syntactically equivalent to itself, yet it may be dynamically impure and generate different value components each time it is evaluated. Since static equivalence in the presence of identity types is a strict generalization of O'Caml's syntactic equivalence, it does not imply dynamic equivalence either. In preparing the language design of this thesis, I decided to dispense with identity types and the muddled semantics that results from them.

That said, the identity types did have some considerable practical benefits in terms of implementation. First, when comparing the static parts of two modules for equivalence, elaboration invariants guaranteed that the modules would be statically equivalent *if and only if* their identity types were equivalent. This greatly simplified the implementation of the type equivalence subroutine used by the elaborator. Second, since all anonymous structure expressions were elaborated to modules containing sealed submodules, the only projectible modules were those in O'Caml-style named form (*e.g.,* `F(X).A`). As a result, the sizes of types projected from modules remained relatively small. In an implementation of the language defined in this chapter, it may be necessary for the elaborator to insert extra type abbreviations to ensure that types do not blow up in size.

In general, type size and type duplication are serious issues that are easy to overlook when formalizing the language. One notable example of type duplication that I encountered was in the elaboration of `datatype` definitions. In Rule 9.96, the sum type con^{sum} that appears in the types of the `datatype`'s in and out coercions is never bound to a type identifier. Thus, in Rule 9.20, which elaborates `datatype` constructor applications by inlining them, the sum injection in the output of the rule is forced to include a duplicate copy of its target con^{sum} type. This means that the large con^{sum} type corresponding to a large `datatype` will be duplicated everywhere one of its many constructors is used in the program.

The TILT compiler addresses this issue by adding to every `datatype` module/signature a "sum" component specified as transparently equal to the corresponding con^{sum} type, so that subsequent references to con^{sum} can be replaced by references to the `datatype`'s "sum" component. This solution works fine for SML, but causes problems with the use of `datatype` specs in recursively dependent signatures. According to the semantics of this chapter, `datatype` definitions in an rds $\rho(mvar).sig$ are permitted to refer to $mvar^{\mathsf{c}}$ because such references will always emanate from value specifications and thus obey the dynamic-on-static restriction. However, with TILT's addition of transparent sum-type components to `datatype` specs, a reference to $mvar^{\mathsf{c}}$ from a `datatype` definition will imply a reference from its sum-type component, which is *not* dynamic-on-static.

I addressed this problem in the implementation as follows. First, I observed that, in signatures output by the elaborator, the specification of a normal (non-"sum") type component of a signature never refers to any "sum" components. Consequently, it is possible to phase-split every signature *sig* into *three* parts $[\![cvar_1{:}knd_1.\ cvar_2{:}tknd_2.\ con]\!]$, where *sig*'s type components are divided between knd_1, which contains the specifications of its normal type components, and $tknd_2$, which contains the (transparent) specifications of its "sum" type components. As the syntax suggests, $tknd_2$ may refer to $cvar_1$, and *con* may refer to $cvar_1$ and $cvar_2$, but knd_1 may refer to neither.

The upshot is that, if an rds $\rho(mvar^{\mathsf{c}}).sig$ contains static-on-static references from `datatype` specs in *sig* to $mvar^{\mathsf{c}}$, these will always be references from the knd_2 part of *sig* to the knd_1 part. As such, they are fundamentally acyclic dependencies, and it is possible to compile such rds's without requiring equi-recursive types. Unfortunately, in order to phase-split these more flexible rds's, I was forced to modify the entire TILT phase-splitter in order to generate three-part output. Hopefully, in future work, we will be able to use three-part phase-splitting in a more general fashion, *e.g.,* to allow rds's to contain any kind of static-on-static dependency that is fundamentally acyclic, in the way that O'Caml's recursive module extension does.

Finally, while the need to control type duplication resulted in certain complications to my implementation, I would say that overall the benefits of targeting a typed intermediate language outweighed the frustrations. The TILT intermediate language typechecker helped me uncover various small bugs in my implementation, and in one notable instance it pointed to a serious error in an earlier formalization of my $\vdash_{\mathrm{can}}$ judgment for computing canonical implementations of signatures.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

The ML module system stands as a high-water mark of programming language support for data abstraction. I began my work toward this dissertation by investigating how to extend ML with recursive modules, with the goal of enhancing its support for data abstraction even further. In the course of my investigation, I have

- Constructed a unifying account of the ML module system, in which the existing ML dialects can be understood as subsystems that pick and choose different features
- Designed a recursive module extension to this unifying account that addresses methodological problems with current recursive module proposals, encourages the use of data abstraction mechanisms in recursive modules, and is easy to explain to the programmer
- Formalized this design, using Harper and Stone's interpretation of Standard ML as a model for developing a significant portion of the language in the framework of type theory

I have also studied the problem of statically detecting whether recursion, under the backpatching semantics used for recursive modules, is *safe*. I have proposed a promising type system for this purpose that improves considerably on existing approaches and enables more efficient compilation of recursive modules. However, as described in Section 7.6, future work still remains with regard to type inference and type system complexity before my proposal can be feasibly incorporated into ML.

I hope that my unifying conception of ML modules, as well as my thorough analysis of the recursive module problem, may serve as a helpful guide to researchers and programmers alike in their attempts to understand the key issues and choices in the design of the ML module system. Although my thesis culminates in the definition of a new ML dialect, this language is most certainly a work in progress, and should by no means be taken as the final word on ML or its module system. Rather, I hope that this language will serve as a foundation for future work, that its limitations will inspire further evolution of the ML module system, and that its formalization in a type-theoretic framework will encourage others to follow suit.

10.2 Future Work

In addition to the future work on safe recursion discussed in Section 7.6, some interesting avenues for future work include the following:

Separate Compilation of Recursive Modules One major omission of existing recursive module proposals that my language of Part III does not address in its present form is support for separate compilation of mutually recursive modules. ML provides support for separate compilation of normal—*i.e.,* hierarchically-dependent—modules through the functor mechanism. However, as explained in Section 5.2.4, functors are not sufficient to handle separate compilation of *recursive* modules because recursive modules require a special kind of elaboration in order to deal with the double vision problem.

Does this imply that separate compilation of recursive modules is hopeless? Not at all. The primary innovation of my recursive module construct is that it respects data abstraction. If mutually recursive modules are sealed with abstract interfaces, then each module is typechecked in a separate context, which exposes its own implementation but hides the implementation of the other module. It should be straightforward to adapt recursive module elaboration in order to *compile* these modules separately as well, but such a separate compilation mechanism is not encodable in terms of the existing mechanisms of my external language.

As a practical matter, I believe that the most appropriate way to introduce such a mechanism is as part of a compilation management system built on top of ML. Although the Definition of Standard ML does not specify anything regarding compilation management, most implementations provide some facility that enables programs to be split into compilation units and that supports incremental recompilation of those units. TILT additionally provides support for "true" separate compilation—any compilation unit may be compiled independently of its imports, so long as the programmer explicitly specifies the signatures of those imports. Formalizing the semantics of compilation units by elaboration into the ML internal language would be a useful contribution in its own right, and it should be possible in such a semantics to use my recursive module elaboration techniques in order to support recursive dependencies between separate compilation units.

Refining the Total/Partial Distinction In my module framework, the distinction between total and partial functors is sufficiently general to account for the kinds of functors found in the existing variants of the ML module system, but it is easy to imagine more subtle gradations. For example, one may want to write a functor `F` whose body contains two submodules `A` and `B`, wherein `A` contains only uses of basic sealing while `B` contains uses of impure sealing. Under my present semantics, `F` will be deemed partial because of `B`'s impurity, but this is more restrictive than necessary. It would be more accurate to treat `F` as partial with respect to the type components of `B` and total with respect to the type components of `A`.

Another way in which my functor semantics is perhaps overly coarse is that it assumes that the body of a functor depends on the entire argument module. Suppose that the argument `X` of the above functor `F` has itself two submodules `A` and `B`, and that the module `A` in `F`'s body depends only on types projected from `X.A`, not `X.B`. In that case, it might be useful to observe that, when `F` is applied to two argument modules with equivalent `A` components, the `A` components in the result will be equivalent as well, regardless of what the arguments' `B` components were.

It is worth exploring whether either of these refinements to the total/partial distinction can be incorporated into my present framework without introducing semantic complexity disproportionate to their practical utility.

Regaining "Statically Total" Functors It is unfortunate that all existing dialects of the ML module system, including my own, only distinguish between separably total and partial functors, because the "statically total" classification is preferable to both in many cases like the `Set` functor. However, in order to support statically total functors, as well as the related mechanism of inseparable sealing, we appear to require some form of dependent type. For example, we would like to treat the `Set` functor as statically total because it should only return compatible type components when applied to *dynamically* equivalent results. What this means, though, is that the type `Set(X).t` depends on the dynamic components of the argument `X` and is thus a kind of dependent type.

As I explained in Section 2.2.1, the O'Caml functor mechanism is an interesting case. It uses syntactic equivalence to compare functor arguments, which in many (but not all) cases implies dynamic equivalence. Consequently, O'Caml functors behave on many (but not all) arguments as if they were statically total. It remains an open problem whether it is possible to develop a mechanism that mimics the semantics of statically total functors on *all* arguments, without the need for actual dependent types.

Data Abstraction with Multiple Principals ML's sealing mechanism allows the implementor of a module to hide information from the clients of that module, *i.e.,* the rest of the program. While the extensions to ML I have given in this thesis do not fundamentally change this implementor/client model of data abstraction, a useful direction for future work may be to generalize this model. There may be many modules in a program, and the implementor of a module `A` may wish to expose its implementation details to certain modules ($\mathtt{B}_1, \ldots, \mathtt{B}_n$) but not to other modules ($\mathtt{C}_1, \ldots, \mathtt{C}_n$). In ML, this can be awkward to achieve. `A` cannot be sealed right where it is bound because that would hide its implementation from the `B` modules. Instead, `A` and the `B`'s must be defined together as submodules of some larger module `AB`, and then `AB` can be sealed to hide implementation details from the `C` modules. Thus, the intentions of one module's implementor may affect how the whole program is structured.

It is worth exploring whether ML can be extended with a more flexible and explicit notion of implementor (aka "principal" or "agent") that would allow the implementor of one module to specify, in a concise and local manner, how different principals in the program may perceive it. Grossman *et al.* [24] have proposed a technique for syntactic proofs of type abstraction properties in the setting of a λ-calculus with multiple principals. It is not immediately clear, though, how to translate their calculus into a language design, or even what language mechanisms for multi-agent data abstraction are the right ones.

Views In ML, the only abstract types whose elements may be pattern-matched are those introduced by `datatype` definitions. Furthermore, although a `datatype` is an abstract type, it is really "concrete" in the sense that it is isomorphic (via its in and out coercions) to a particular sum type. To remedy this limitation, Wadler [80] proposed the idea of *views*, which allow the programmer to write a non-canonical implementation of a `datatype` signature. A view consists of two transformation functions: one from the `datatype` to the actual implementation type, which is called at applications of the datatype's constructors, and one in the other direction, which is invoked during pattern matching.

Views afford programmers the convenience of pattern matching, while preserving their ability to hide the identity of the implementation type. As such, they would constitute a real extension to ML's support for data abstraction. Okasaki has proposed a variant of Wadler's views in the context of SML [60], but only informally. It would be useful to formalize his (or perhaps some other) view extension using the Harper-Stone framework.

Type Classes As I explained at the beginning of Chapter 9, my language deprecates the SML features of overloaded equality and equality types because I believe they are more trouble than they are worth. A general mechanism for overloading, though, would be a worthwhile extension to ML. Originally proposed by Wadler and Blott [81] as a way of generalizing ML's equality types, *type classes* enable overloading of arbitrary user-defined functions and have become one of the most popular features of the Haskell language [25].

Although Haskell type classes, like ML modules, provide support for code reuse, type classes are no substitute for a module system, and vice versa. Type classes generalize type inference, whereas modules provide program structure and enforce data abstraction. An important (and, to my knowledge, unexplored) direction for future work is to understand better how these features relate to each other and how they might interact if they were both provided by a single language.

Bibliography

[1] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.

[2] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[3] Davide Ancona, Sonia Fagorzi, Eugenio Moggi, and Elena Zucca. Mixin modules and computational effects. In *International Colloquium on Automata, Languages, and Programming*, Eindhoven, The Netherlands, 2003.

[4] Davide Ancona and Elena Zucca. A primitive calculus for module systems. In *International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 62–79. Springer-Verlag, 1999.

[5] Gerard Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 14(3):263–315, May 2004.

[6] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, Atlanta, GA, 1999.

[7] Pierre Crégut and David B. MacQueen. An implementation of higher-order functors. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Standard ML and its Applications*.

[8] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, Albuquerque, NM, 1982.

[9] Derek Dreyer. Moscow ML's higher-order modules are unsound. Posted to the TYPES electronic forum, September 2002.

[10] Derek Dreyer. A type system for well-founded recursion. In *Thirty-First ACM Symposium on Principles of Programming Languages (POPL)*, pages 293–305, Venice, Italy, January 2004.

[11] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules (expanded version). Technical Report CMU-CS-02-122R, School of Computer Science, Carnegie Mellon University, December 2002.

[12] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Thirtieth ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, LA, January 2003.

[13] Derek Dreyer, Robert Harper, and Karl Crary. A type system for well-founded recursion. Technical Report CMU-CS-03-163, Carnegie Mellon University, July 2003.

[14] Dominic Duggan. Type-safe linking with recursive DLL's and shared libraries. *ACM Transactions on Programming Languages and Systems*, 24(6):711–804, November 2002.

[15] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, June 1996.

[16] Dominic Duggan and Constantinos Sourelis. Parameterized modules, recursive modules, and mixin modules. In *1998 ACM SIGPLAN Workshop on ML*, pages 87–96, Baltimore, Maryland, September 1998.

[17] Levent Erkök and John Launchbury. Recursive monadic bindings. In *2000 International Conference on Functional Programming*, pages 174–185, Paris, France, 2000.

[18] Levent Erkök and John Launchbury. A recursive do for Haskell. In *2002 Haskell Workshop*, October 2002.

[19] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[20] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, Montreal, Canada, June 1998.

[21] Daniel P. Friedman and Amr Sabry. Recursion is a computational effect. Technical Report TR546, Indiana University, December 2000.

[22] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998.

[23] John Greiner. Weak polymorphism can be sound. *Journal of Functional Programming*, 6(1):111–141, 1996.

[24] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, November 2000.

[25] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[26] Robert Harper. Programming in Standard ML. `http://www.cs.cmu.edu/~rwh/introsml/`.

[27] Robert Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):463–469, 1999.

[28] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[29] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[30] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[31] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[32] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1997.

[33] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[34] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[35] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *2002 European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 6–20, 2002.

[36] Tom Hirschowitz, Xavier Leroy, and J.B. Wells. Compilation of extended recursion in call-by-value functional languages. In *Principles and Practice of Declarative Programming (PPDP)*, pages 160–171, Uppsala, Sweden, August 2003.

[37] Mark P. Jones. Using parameterized signatures to express modular structure. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 68–78, St. Petersburg Beach, FL, 1996.

[38] Richard Kelsey, William Clinger, and Jonathan Rees (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), September 1998.

[39] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[40] Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, 1976.

[41] Xavier Leroy. The Objective Caml system: Documentation and user's manual. Available online at `http://caml.inria.fr/ocaml/htmlman/`.

[42] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 1994. ACM.

[43] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of POPL '95: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, CA, January 1995.

[44] Xavier Leroy. A proposal for recursive modules in Objective Caml, May 2003. Available online at: `http://cristal.inria.fr/~xleroy/publi/recursive-modules-note.pdf`.

[45] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1996.

[46] David MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 198–207, Austin, TX, 1984.

[47] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[48] David MacQueen. Adaptation in HOT languages: Comparing polymorphism, modules, and objects. In C. A. R. Hoare, M. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*. IOS Press, Amsterdam, 2001.

[49] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald T. Sannella, editor, *Programming Languages and Systems — ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

[50] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1990.

[51] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[52] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[53] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[54] Eugenio Moggi and Amr Sabry. An abstract monadic semantics for value recursion. In *2003 Workshop on Fixed Points in Computer Science*, April 2003.

[55] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. Gordon and A. Pitts, editors, *Higher Order Operational Techniques in Semantics*. Newton Institute, Cambridge University Press, 1997.

[56] Moscow ML. `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[57] Aleksandar Nanevski. Meta-programming with names and necessity. In *2002 International Conference on Functional Programming*, pages 206–217, Pittsburgh, PA, 2002. A significant revision is available as a technical report CMU-CS-02-123R, Carnegie Mellon University.

[58] Aleksandar Nanevski. *Functional Programming with Names and Necessity*. PhD thesis, Carnegie Mellon University, June 2004. Available as Technical Report CMU-CS-07-9254872.

[59] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[60] Chris Okasaki. Views for Standard ML. In *1998 ACM SIGPLAN Workshop on Standard ML*, pages 14–23, Baltimore, MD, September 1998.

[61] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University, December 2000.

[62] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[63] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999.

[64] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer, 2000.

[65] Claudio V. Russo. *Types for Modules*. PhD thesis, Edinburgh University, Edinburgh, Scotland, 1998. LFCS Thesis ECS–LFCS–98–389.

[66] Claudio V. Russo. Recursive structures for Standard ML. In *International Conference on Functional Programming*, pages 50–61, Florence, Italy, September 2001.

[67] Chung-chieh Shan. Higher-order modules in System F_ω and Haskell. Unpublished manuscript, available from the author's website.

[68] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

[69] Zhong Shao. Transparent modules with fully syntactic signatures. In *International Conference on Functional Programming*, pages 220–232, Paris, France, September 1999.

[70] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. In *1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, CA, 1995.

[71] Standard ML of New Jersey. `http://www.smlnj.org`.

[72] Christopher A. Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 2000.

[73] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. Technical Report CMU-CS-99-155, School of Computer Science, Carnegie Mellon University, September 1999.

[74] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 214–227, Boston, January 2000.

[75] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 2005. To appear.

[76] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[77] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[78] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988.

[79] Joseph C. Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. Typed compilation of recursive datatypes. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, New Orleans, LA, January 2003.

[80] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 307–313, 1987.

[81] Philip Wadler and Stephen Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.

[82] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.