Translucent Sums: A Foundation for Higher-Order Module Systems

Mark Lillibridge

May, 1997

CMU–CS–97–122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Thesis Committee:
Robert Harper, Chair
Peter Lee
John Reynolds
Luca Cardelli, DEC SRC

This research was sponsored by the Air Force Materiel Command (AFMC) and the Defense Advanced Research Projects Agency (DARPA) under contract number, F19628-95-C-0050. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

Abstract

The ease of understanding, maintaining, and developing a large program depends crucially on how it is divided up into modules. The possible ways a program can be divided are constrained by the available modular programming facilities ("module system") of the programming language being used. Experience with the Standard-ML module system has shown the usefulness of functions mapping modules to modules and modules with module subcomponents. For example, functions over modules permit abstract data types (ADTs) to be parameterized by other ADTs, and submodules permit modules to be organized hierarchically. Module systems with such facilities are called higher-order, by analogy with higher-order functions.

Previous higher-order module systems can be classified as either *opaque* or *transparent*. Opaque systems totally obscure information about the identity of type components of modules, often resulting in overly abstract types. This loss of type identities precludes most interesting uses of higher-order features. Transparent systems, on the other hand, completely reveal type identities by inspecting module implementations, which subverts data abstraction and prevents separate compilation.

In this dissertation, I describe a novel approach that avoids these problems based on a new type-theoretic concept, the translucent sum. A translucent sum is a kind of weak sum whose type can optionally specify the identity of its type components. Under my approach type identities are not determined by inspecting module implementations, permitting separate compilation. By default, module operations reveal the connections between the types in their input and output modules. However, these connections can be obscured using type coercion. This controlled visibility permits data abstraction where desired without limiting the uses of higher-order features. My approach also allows modules to be treated as first-class values, a potentially valuable feature.

In order to lay out the groundwork for my new approach to designing higher-order module systems and to demonstrate its utility, I develop in complete detail a kernel system using my approach. I establish formally the theoretical properties of the system, including soundness even in the presence of side effects. I also show how the system may be effectively type checked.

Contents

Chapter 1

Introduction

1.1 Overview

Many programming languages have a collection of facilities for building modular programs. These collections of facilities, called *module systems*, play an important role in programming languages, especially with regard to programming-in-the-large. This dissertation concerns a new approach to designing module systems for statically-typed programming languages. The approach promises a substantially more powerful module system than those built with previous approaches, while at the same time eliminating the problems associated with the previous approaches.

Traditional module systems, such as the one provided by Modula-2 [56], are *first-order*, allowing only trivial manipulations of modules. Some newer programming languages provide *higher-order* module systems. Higher-order module systems, unlike first-order ones, permit the non-trivial manipulation of modules within the language. In particular, they at least permit functions mapping modules to modules and may provide other higher-order features such as modules containing modules as subcomponents and modules as first-class values. Experience with the Standard-ML module system has shown the usefulness of functions mapping modules to modules and modules with module subcomponents. For example, functions over modules permit abstract data types (ADTs) to be parameterized by other ADTs, and submodules permit modules to be organized hierarchically. Modules as first-class values is also likely to be useful because it permits choosing ADT implementations at runtime.

Previous higher-order module systems can be classified as either *opaque* or *transparent*. Opaque systems totally obscure information about the identity of type components of modules, often resulting in overly abstract types. This loss of type identities precludes most interesting uses of higher-order features. Transparent systems, on the other hand, completely reveal type identities by inspecting module implementations, which subverts data abstraction and prevents separate compilation. Unlike opaque systems, transparent systems cannot support modules as first-class values. My thesis is as follows:

> By basing a module system on a new type-theoretic concept, namely a new kind of weak sum whose type can contain both transparent and opaque type declarations called a *translucent sum*, it is possible to obtain a higher-order module system with the advantages of both the opaque and the transparent approaches, but none of their disadvantages.

In order to demonstrate this thesis, I design a new programming language with a higher-order module system based on translucent sums. The language I have created is a kernel system that contains only the features relevant to module system design; considerable extension would be required to make it into a real programming language. The design of this kernel system and the proofs required to establish formally its properties form the core of this dissertation. Included are a proof of the system's soundness and effective procedures for type checking its programs.

This dissertation is divided into three parts. The first part introduces the problem and provides the needed background on module systems (this chapter), explains the previous approaches to designing higher-order module systems (Chapter 2), and describes my approach using translucent sums (Chapters 3 and 4). The second part, comprising Chapters 5 to 12, is devoted to the kernel system and its associated proofs. An overview of this part can be found in Chapter 5; descriptions of this part's other chapters can be found in Section 5.10. Finally, the third part discusses possible extensions to the kernel system (Chapter 13), explains related work (Chapter 14), and summarizes this dissertation's results (Chapter 15).

1.2 Module Systems

A *module* binds a set of values and types to names. For example, we might have

```
M = module
        val  x = 3;
        val  y = true;
        type T = int;
    end;
```

This code creates a new module with three components, called `x`, `y`, and `T`. The components `x` and `y` are value components that are bound to the values `3` and `true` respectively; the component `T`, by contrast, is a type component that is bound to the type `int`. Once this module is created, it is then bound to the name `M`; this binding will allow us to refer to the new modules' components as `M.x`, `M.y`, and `M.T` later on. Although in this example, we named our new module, this is not required. Sometimes, for example, we may want to create a new module then immediately apply a function to it; naming the new module serves no purpose in that case.

Because I am dealing with statically typed languages, modules will have "types", called *interfaces*. For example, the module `M` matches the following interface:

```
M : interface
        val  x:int;
        val  y:bool;
        type T;
    end;
```

This interface specifies that `M` has two value components, `x` with type `int` and `y` with type `bool`, and one type component, `T`.[1] The relationship between modules and interfaces is many-to-many: Many modules may match the same interface and a given module may match many interfaces.

A programming language's collection of facilities for building modular programs is its module system. I classify module systems into two types, depending on what sort of facilities they have. *First-order* module systems have only the trivial module facilities discussed so far: module creation, module naming, and module-component extraction (`M.x`). Most traditional module systems are of this type. Examples include Ada [52], CLU [36], C [29], C++ [53], and Modula-2 [56].

Higher-order module systems, by contrast, have non-trivial module manipulation facilities. I shall be concerned in this dissertation primarily with three such facilities: *functors*, *submodules*, and modules as first-class values. Functors are functions mapping modules to modules. For example, we might define a rectangle ADT parameterized by a point ADT using a functor:

```
MkRect = functor(p:POINT):RECT
             module
                 type T = p.T * p.T;
                 ...
              end;
```

The advantage of doing this is that we can create several rectangle ADTs from the same code using different point ADTs. For example, if we have point ADTs `CartPoint` and `PolarPoint`, we can create a rectangle ADT based on Cartesian points with `CartRect`

[1] In my kernel system I deal with a more complicated version of modules that assigns "types", called *kinds*, to types; I have simplified things here.

`= MkRect(CartPoint)` and a rectangle ADT based on polar points with `PolarRect = MkRect(PolarPoint)`.

Submodules are modules contained as components within other modules; they allow packaging up a series of modules into a single unit. For example, if we had several search-tree modules, we could package them up into a single module:

```
SearchTree = module
                 mod Binary   = BinarySearchTree;
                 mod RedBlack = RedBlackSearchTree;
                 mod BTree    = BTree;
             end;
```

`SearchTree`, in turn, could be included as part of a larger library algorithms module. By organizing modules in this way, a hierarchical namespace can be obtained; such namespaces are easier for programmers to deal with than flat namespaces. Because most module systems do not consider modules to be ordinary values, a different keyword (**mod**) is needed in general here to specify that these are module components, not value components.

Modules as first-class values refers to the ability to treat modules as if they were ordinary values of the programming language. This means that anything that can be done with a ordinary value can be done with a module. Thus, in languages with modules as first-class values, modules can be passed to or returned by functions, stored in variables, or selected using conditional statements. In such languages, the **mod** keyword is just syntactic sugar for the **val** keyword.

One useful way to use this facility is to select the most efficient implementation for an ADT at runtime. For example, the best implementation for a dictionary depends on the number of items it will contain; the following code sets `Dictionary` at runtime to use an implementation that is efficient for `n` items:

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

1.3 Benefits of Higher-Order Module Systems

Higher-order module systems offer many benefits over first-order module systems. I have already mentioned some benefits (the ability to parameterize with respect to ADTs, the ability to have a hierarchical namespace, and the ability to choose ADT implementations at runtime), but the greatest benefit of higher-order module systems is that they allow programs to be organized better. Program organization is crucial to ensuring that large programs are easy to develop, understand, and maintain.

The most well-known programming language with a higher-order module system is Standard ML (SML) [21]. SML provides functors and submodules, but treats modules as

normal:	mobile:	local:
	TCP	
TCP	*MOBILE-IP*	TCP
IP	IP	*glue*
ETH	ETH	ETH
DEV	DEV	DEV

Figure 1.1: TCP/IP protocol organization

second-class values. The extensive experience of the SML community with functors and submodules has established their value as program-organization tools. Modules as first-class values shows promise as a valuable addition to module system toolkits, but since this facility has never been implemented together with the other higher-order facilities, experience about its value is lacking.

It is hard to find good examples of why better organization matters: Any small section of a program can usually be rewritten so that it no longer uses higher-order module system features at the cost of making the rest of the program more complicated. Nonetheless, I am now going to present two examples that I feel convey some of the flavor of how higher-order modules systems are used in practice.

My first example comes from Carnegie Mellon's Fox Project's implementation of the TCP/IP protocol suite, the FoxNet [6, 5, 4], in SML. The TCP/IP protocol suite is organized modularly in the form of a protocol stack (see Figure 1.1). The leftmost stack in this figure shows the organization of normal TCP/IP. Normal TCP/IP is built on a base device driver protocol (DEV), upon which additional protocol layers (ETH, IP, and TCP in turn) are placed. Each of the additional protocol layers takes in the lower level protocol implemented by the underlying stack and produces a new protocol with additional functionality. For example, the IP layer adds the ability to route between different nets.

What is particularly nice about this organization is that the protocol interfaces of the layers are (mostly) identical, allowing protocol layers to be mixed and matched in order to generate many kinds of functionality. For example, the middle stack in Figure 1.1 represents a configuration for mobile TCP/IP, where in an extra protocol layer, MOBILE-IP, has been added between the TCP and IP layers in order to provide an extra layer of indirection between virtual and physical IP addresses; mobile TCP/IP allows computers to move around physically without losing connections.

Alternatively, the rightmost stack in the figure represents a configuration for local TCP/IP, where the IP layer has been dropped. Because the resulting protocol lacks any inter-net routing ability, it can be used only between hosts on the same local net; however, it will be faster for this purpose because it does less processing and uses smaller packets. (A small glue layer, labeled *glue*, is necessary between the TCP and ETH layers because the protocol interfaces these layers use are slightly different in practice.)

By using functors, FoxNet is able to directly capture this modular organization. Idealized code for doing this might look like the following:

```
DEV : PROTOCOL = module ... end;

ETH = functor (lower:PROTOCOL):PROTOCOL
        module ... end;
...
```

Here, the base device layer `DEV` is implemented as a normal module matching the PROTOCOL interface and the other layers are implemented as functors mapping a PROTOCOL to a PROTOCOL. Using this code, the normal and local TCP/IP protocol stacks could then be built up by applying the reverent layer functors in the correct order:

```
Normal = TCP(IP(ETH(DEV)));
Local  = TCP(Glue(ETH(DEV)));
```

In a language with modules as first-class values, we could choose which of these protocol stacks to use at runtime for maximum efficiency:

```
P = if on_local_net() then Local else Normal;
```

Note that the resulting implementation is completely type safe and does not require any runtime type checks. A similar-looking object-oriented programming (OOP) program could be written where protocols and packets are objects. (Each protocol contains a packet type, which is the type of packets belonging to that protocol.). The resulting program would be much slower though because it would require frequent runtime type checks. (Higher-level–protocol packets have extra fields not present in lower-level–protocol packets; runtime type checks must be done before those fields can be accessed because first-order module systems cannot statically distinguish between the different packet types.)

1.4 Example: B-Trees

My second example involves an ordered dictionary implementation. I have chosen this example carefully and will be using it throughout the first part of my dissertation because it illustrates well the differences between the various approaches to building higher-order module systems.

An ordered dictionary is a dictionary whose entries are ordered by an ordering on its keys. The classic example of an ordered dictionary is the white pages telephone directory: You can look up phone numbers by name as well as examine entries in alphabetical order.

It order to define the notion of an ordered dictionary, it is first necessary to define the notion of an *ordered type*:

```
ORDERED = interface type T;
                    val   cmp: T*T -> int;
          end;
```

Here, I have defined an ordered type to be a module containing a type `T`[2], the type in question, together with a comparison function `cmp` that compares two values of type `T`, returning an integer that indicates how they compare. (That is, `cmp(x,y)<0` if `x < y`, `cmp(x,y)=0` if `x = y`, and `cmp(x,y)>0` if `x > y`.)

Using this definition, the notion of an ordered dictionary ADT can then be defined:

```
ORDERED_DICTIONARY =
  interface
     mod Key:ORDERED;
     type elem;
     type T;

     val new:    unit -> T;
     val define: T * Key.T * elem -> unit;
     val first:  T -> Key.T * elem;
     ...
  end;
```

Here, `Key` is the ordered type of keys, `elem` is the type of the elements looked up, and `T` is the type of ordered dictionaries managed by the ADT. The functions `new`, `define`, `first`, and so on, are the operations on ordered dictionaries. For example, `new()` creates a new empty ordered dictionary (the type `unit` contains only one value, denoted `()`), `define(d,k,e)` adds the binding of key `k` to element `e` to the ordered dictionary `d`, and `first(d)` either returns the first binding in the dictionary or raises the exception `fail`.

An example implementation of an ordered dictionary ADT might then be obtained using B-Trees as follows:

[2]By convention, the primary type of a module is called `T`.

```
MkBTree =
  functor (K:ORDERED, E:interface type T; end)
             :ORDERED_DICTIONARY
    module
       mod Key = K;
       type elem = E.T;
       type T =  btree(Key.T,elem);

       fun new() = ...
       fun define(dict:T, key:Key.T, value:elem) = ...
       ...
    end;
```

I have parameterized the implementation with respect to which ordered type to use for keys (`K`) and which type to use for elements (`E.T`).[3] The implementation first makes copies of the key and element information so that future clients will have access to this information; second, establishes the representation type it will use, which I have abbreviated here as *btree(*`Key.T`*,*`elem`*)* so that I can refer to it later easily; and, finally, third, it provides code for each of the required ordered dictionary operations (not shown).

Because the behavior of `MkBTree` varies depending on which approach to designing higher-order module systems is being used, I shall defer further discussion of `MkBTree` until I have described the approaches in more detail.

[3] For technical reasons, it is necessary to wrap up the element type in a module in order to pass it to a functor; this fact is not important here.

Chapter 2

Previous Approaches

In order to motivate my approach to designing higher-order module systems, I shall now describe the previous approaches to designing higher-order module systems. The previous approaches can be classified as either "opaque" or "transparent", depending on how they treat module and functor boundaries.

2.1 The Opaque Approach

Under the opaque approach, module and functor boundaries are opaque, allowing no information about the identity of type components to pass through. This is exactly what we want when we build an ADT. For example, consider the following definition of an integer stack ADT:

```
IntStack = module
               type T = ref(list(int));
               fun  new() = new [];
               fun  push(s:T, e:int) = ...
               ...
           end;
```

The identity of the integer stack type (`T`) is visible inside the module, allowing us to implement its operations using the appropriate primitives on its representation type, but its identity is obscured outside the module: `IntStack.T` is an abstract type. Thus, the user of this ADT will be able to create and manipulate integer stacks using code like `IntStack.pop(IntStack.push(IntStack.new(), 3))`, which returns `3`, but will not be able to depend in any way on which representation it uses.

Examples of programming languages which take this approach are John Mitchell and Gordon Plotkin's SOL [43] and Luca Cardelli's Quest [10]. This approach provides data abstraction at the module level, as we have just seen. It also allows *separate compilation*, the ability to type check and compile individual module implementations using only the interfaces of the modules they reference, because modules depend only on interfaces, not on implementations. This approach is also compatible with modules as first-class values because it makes no assumptions about type-component identities.

Unfortunately, because this approach obscures the identity of *all* type components, it precludes most interesting uses of higher-order features. Consider the B-Tree example I introduced in Section 1.4. Suppose we try to use it to create a telephone directory. To keep things simple, let us use the type `string` as both the key and element type. We can build an ordered type "`string`, ordered alphabetically" by wrapping up `string` with the appropriate `string` comparison operator:

```
StringOrd = module
                type T   = string;
                type cmp = ...;
            end;
```

A simple functor application then suffices to create an ordered dictionary using this ordered type for keys and `string`'s for elements:

```
D = MkBTree(StringOrd, module type T=string; end);
```

As a test of our new ADT, we can try creating a new telephone directory and then adding an entry to it:

```
directory = D.new();

D.define(directory, "John Smith", "123-4567");
```

Unfortunately, the second line above fails to type check under the opaque approach. The reason is that `D.define` has the type

$$\texttt{D.T} * \texttt{D.Key.T} * \texttt{D.elem} -> \texttt{unit}$$

which specifies that `D.define`'s second and third arguments are of type `D.Key.T` and `D.elem` respectively; because the opaque approach treats all type components abstractly outside their module of definition, these two types are not compatible with `string`, the type of the passed second and third arguments. This problem renders the B-Tree implementation unusable under the opaque approach.

The problem with this example is an instance of a more general problem: There is no way in the opaque approach to build modules "containing" other types; we can only build modules containing unrelated new types (e.g., `StringOrd.T` $\neq$ `string`). Hence, modules cannot export type abbreviations; for example, we cannot make `Complex.T` be an abbreviation for the type `Real*Real`. Also, we cannot extend ADTs by adding a few new operations to an existing ADT.[1] Another consequence is that many of the more useful idioms using higher-order features are inexpressible. Indeed, David MacQueen argued in a 1986 paper [37] that this problem is severe enough that the opaque approach should be abandoned in favor of a transparent one.

2.2 The Transparent Approach

Unlike in the opaque approach, under the transparent approach, module and functor boundaries are transparent, allowing all information about the identities of type components to pass through them; the transparent approach does this by inspecting module implementations to determine the actual identity of type components. For example, consider again the `IntStack` example:

```
IntStack = module
               type T = ref(list(int));
               fun  new() = new [];
               fun  push(s:T, e:int) = ...
               ...
           end;
```

This time, the identity of the integer stack type is visible both inside and outside of the module: We know that `IntStack.T = ref(list(int))` and can thus manipulate integer stacks using reference and list primitives. The identity of `IntStack.T` is computed by substituting in its implementation and then reducing the resulting expression:

```
IntStack.T =
(module type T=ref(list(int)); ... end).T =
ref(list(int))
```

Examples of programming languages that take this approach are David MacQueen's DL [37]; Robert Harper and John Mitchell's XML [41]; and Robert Harper, John Mitchell, and Eugenio Moggi's λ^{ML} [22]. As we have just seen, the transparent approach does not provide data abstraction at the module level. It is possible, however, to obtain a more limited kind of data abstraction, which I call *closed* abstraction, under the transparent approach.

---

[1] For example, we cannot create a new module `ExtendedString` from a built-in `String` ADT module by first copying the `String` type and operations then adding a few new operations, and have extended strings be interchangeable with the built-in strings.

Closed abstraction is abstraction limited to a specific closed scope. For example, we might have an abstract-in construct that allows us to hold our integer-stack module abstract in a particular piece of code (M below):

```
INTSTACK = interface
               type T;
               val  new : unit -> T;
               ...
           end;

abstract IntStack:INTSTACK = module
                                 type T = ref(list(int));
                                 fun  new() = new [];
                                  ...
                               end
  in
    M
  end;
```

Here `IntStack` is held abstract inside M; that is, only the information available in the interface given for it (`INTSTACK`) is visible inside M. Thus, the fact that `IntStack.T` is a type is visible but not the fact that `IntStack.T` is equal to `ref(list(int))`. Because `IntStack` is only visible in M, this construct provides only closed abstraction.

The abstract-in construct can be desugared in the transparent approach into a functor application:

```
(functor (IntStack:INTSTACK) M)
  (module
     type T = ref(list(int));
     fun  new() = new [];
     ...
   end);
```

Because in general it is not possible to determine the code for functor arguments at compile time, the transparent-system type checker checks M using only the information available in interface for `IntStack`, giving the desired behavior.

By contrast, in *open* abstraction, which is provided by my approach and the opaque approach but not by the transparent approach, the scope of the abstraction is not limited to a single existing scope. For example, using an open-abstraction construct (denoted by **abs** here), we might have the following code:

```
S = module
      abs IntStack:INTSTACK = module
                                type T = ref(list(int));
                                fun  new() = new [];
                                ...
                              end;
      ...
    end
...
```

Here our integer-stack module is held abstract both inside the module `S` (under the name `IntStack`) and outside the module `S` (under the name `S.IntStack`). If `S` is at the top level of a program then the later scope may include modules complied separately from `S` at a later time. Closed abstraction cannot handle such modules because it requires that all code that uses an ADT be available at the time the ADT is written. Because of this important limitation, I do not consider systems that support only closed abstraction to provide abstraction at the module level. Note that under the opaque approach **abs** is just a synonym for **mod** because abstraction is automatic.

Because the transparent approach requires access to the implementations of all the modules a module refers to in order to compile that module (in order to determine the identities of type components from those modules), it cannot provide separate compilation. Also, the transparent approach is incompatible with treating modules as first-values because there is no way to determine the actual identity of a type when it depends on information available only at runtime.[2]

However, the transparent approach does allow modules to "contain" other types. This fact allows most of the interesting use of higher-order features to be used under the transparent approach. For example, if we try and build a telephone directory using `MkBTree` like we did before (see Section 2.1), no type errors occur this time because under the transparent approach, `D.define` has the type *btree(*`string`,`string`*)* `* string * string -> unit`. Accordingly, the resulting telephone dictionary is fully usable, but its implementation is exposed (note the first argument's type), which is undesirable from the point of view of information hiding.

2.3 Summary

Summarizing, there are two main previous approaches to designing higher-order module systems, the opaque approach and the transparent approach. The opaque approach has

[2]More precisely, unsoundness results from this combination in the presence of side effects; the unsoundness example in Section 4.5 shows this fact because it is well typed under the transparent approach extended with first-class modules but causes a type error at runtime.

the following advantages:

- Provides module-level data abstraction
- Supports separate compilation
- Compatible with modules as first-class values

Its disadvantages include the absence of all of the following:

- Modules can "contain" other types
- Modules may export type abbreviations
- Many of the more useful higher-order idioms work

The transparent approach, on the other hand, has the opposite advantages and disadvantages; it lacks the opaque approach's advantages but it removes the opaque approach's disadvantages. Both approaches' set of disadvantages are serious enough to result in their each building unsatisfactory module systems.

Chapter 3

My Approach

In this chapter I introduce the basic ideas of my approach to designing higher-order module systems, contrast my approach with the previous ones, and explore my approach's abilities. In the next chapter (Chapter 4), I explain in more detail the technical machinery behind my approach.

3.1 Module and Functor Boundaries

Under my approach, which I call the *translucent* approach, module and functor boundaries are "translucent," being opaque in some places and transparent elsewhere. Where the boundaries are opaque is controlled by the programmer using interfaces. Each boundary has an associated interface, which specifies which type component's identity information can be seen through the boundary.

In order to allow this specification, interfaces in my approach are extended to allow specifying the identity of type components. For example, the following interface for a hash table ADT specifies the type identity of the element type (`elem`) but not that of hash tables (`T`):

```
HashTable : interface type elem = int;
                      type T;
                      ...
            end
```

Using this ability, the programmer specifies boundary properties by giving, in the interface associated with a boundary, the type identities of only those components she wishes the boundary to be transparent to; all other type components will be treated opaquely. Thus, in the `HashTable` example above, the boundary will be transparent for the `elem` component and opaque for the `T` component.

Because the programmer is required to provide the actual type identity of transparently-treated type components in the interface (the type checker checks that the type identities provided in the interface are correct when the interface is assigned to a module), the type checker does not need to inspect the implementation of modules to determine the identity of their type components; it can just believe the module's interface. This fact means that modules depend on only the interfaces of other modules; my approach is thus able to support separate compilation without any trouble.

3.2 Default Interfaces

If the programmer does not supply an interface for a *simple* module (**module** ... **end**), then the type checker infers a fully transparent interface by inspecting the module's implementation. For example, under my approach, `StringOrd`, which we bound to the result of evaluating **module type** `T = string;` **type** `cmp = ...;` **end**, would have been assigned the following interface:

interface type `T = string;` **val** `cmp: T*T -> int;` **end**

If we compare this interface to the original opaque `ORDERED` interface (repeated below), we see that the new interface differs only in the addition of the type identity `T=string`:

interface type `T;` **val** `cmp: T*T -> int;` **end**

By using a shorthand notation due to Xavier Leroy [31], we can take advantage of this fact to express the new interface as `ORDERED` **with** `T=string`. Leroy's with notation is a purely syntactic shorthand that expresses the result of adding information about type identities to an existing interface. A discussion of exactly how the with notation works can be found in Subsection 13.7.4.

The situation for non-simple module expressions without interfaces is more complicated, particularly because modules under my approach are treated as first-class values. Briefly, by the use of some type rules, an interface that is as transparent as possible given soundness and reasonable compile-time–information constraints[1] is inferred. I shall give an example of how this can occur later in Section 3.7. Code that does not involve using modules as first-class values, *higher-order functors* (functors that take or return other functors as arguments), or type coercions will be given a fully transparent interface [34]. This category includes the majority of code written in SML.

[1]In particular, the type checker makes no attempt to determine the run-time branching of conditionals, instead assuming that they could branch either way at any time.

3.3 Subtyping

Subtyping in my system permits forgetting type identities; for example, `ORDERED` **with** `T=int` is a subtype of `ORDERED`. This fact means that the programmer can use type coercions (M `<:` **interface** ... **end**) to increase opacity where and when desired. For example, the following code creates an initially transparent `HashTable` module then makes opaque just the `T` type component:

```
HashTable = (module type elem = int;
                    type T = (string*elem) array;
                    ...
             end
           <: interface type elem = int;
                     type T;
                     ...
            end);
```

This technique provides module-level data abstraction under my approach.

Thus, in summary, under the translucent approach, the default is full transparency; opacity results either from using modules in a first-class manner (to prevent unsoundness) or from using an explicit type coercion (to obtain data abstraction).

3.4 The B-Tree Example

To get a feel for how translucency works, let us consider again the B-Tree example from Section 1.4. What happens if we try to build a telephone directory using `MkBTree` under the translucent approach? If we examine the original code for `MkBTree` (partially copied below), we see that it assigns the interface `ORDERED_DICTIONARY` to the result of applying `MkBTree`:

```
MkBTree =
  functor (K:ORDERED, E:interface type T; end)
            :ORDERED_DICTIONARY
    module
       mod Key = K;
       type elem = E.T;
       type T =  btree(Key.T,elem);
       ...
    end;
```

Because `ORDERED_DICTIONARY` is completely opaque (it specifies no type identities), this means that `MkBTree`'s boundary under the translucent approach will be completely opaque as well; consequently, `MkBTree` will act the same as it did under the opaque approach: If we try to build a telephone directory like we did before in Section 2.1, abstract operation-argument types will result in type errors that prevent us from using the resulting telephone directory.

What happens if we remove the explicit result interface (`:ORDERED_DIRECTORY`) from the `MkBTree` code? In that case, the type checker will infer the following totally-transparent result interface:

```
ORDERED_DICTIONARY with Key.T = K.T
                   with elem  = E.T
                   with T     = btree(K.T,E.T)
```

(Note that the body of the `MkBTree` functor is a simple module.)

With this interface, `MkBTree`'s boundary will be completely transparent; consequently, this version of `MkBTree` will act the same as the old version of `MkBTree` did under the transparent approach: If we try to build a telephone directory, the resulting dictionary will be fully usable, but its implementation will be exposed to the user.

Both versions of `MkBTree` are unsatisfactory; what we really need is for `Key.T` and `elem` to be transparent (so that the resulting dictionary will be usable) and for `T` to be opaque (so that `MkBTree`'s implementation is hidden). We can achieve this by assigning the following translucent interface as the return interface for `MkBTree`:

```
ORDERED_DICTIONARY with Key.T = K.T
                   with elem  = E.T
```

This interface gives `MkBTree` a translucent boundary that is transparent for the `Key.T` and `elem` type components but opaque for the `T` type component.

What happens if we try and create a telephone directory using this new version of `MkBTree` as before (code repeated below)?

```
D = MkBTree(StringOrd, module type T=string; end);

directory = D.new();

D.define(directory, "John Smith", "123-4567");
```

Because of the translucent boundary, `D.define` will have the type `D.T * string * string -> unit`, where `D.T` is an abstract type. Accordingly, there is no problem using the resulting telephone directory and the implementation of `MkBTree` is hidden as desired. Note that only my approach is thus able to handle the B-Tree example in a satisfactory manner.

3.5 The Recipe

My approach is based on type theory and the λ-calculus. In it, modules are a particular sort of value and interfaces are a particular sort of type. This choice yields a simple and uniform design.

My "recipe" for building translucent higher-module systems is as follows:

- Start with a base λ-calculus (to model the core language); I use Girard's F_ω [15] in my Kernel system
- Add *translucent sums* (to module modules)
- Add dependent functions (to model functors); *dependent* functions are needed because the result type of a functor can depend on its argument
- Add a notion of subtyping (to model implementation-interface matching)

The key component here is the new type theoretical construct I call a translucent sum; the other components of the recipe are, with the exception of one minor change to the behavior of dependent functions (see the next chapter), standard components drawn from the extensive type-theory literature.

Indeed, variations of this recipe can be used to build opaque or transparent higher-order module systems: To get an opaque system, for example, weak sums (also called existentials) with the dot notation elimination form (discussed in the next chapter) should be substituted for translucent sums in the recipe. To get a transparent system, on the other hand, strong sums should be substituted instead. It will also be necessary in the transparent case to stratify the system, separating modules, functors, and their "types" from ordinary terms and types; this stratification ensures soundness at the cost of making modules and functors second-class values.

For both of these variations, in order to handle submodules, dependent sums should either be added in addition or their functionality should be merged into the weak or opaque sum construct. (Translucent sums already include the functionality of dependent sums.)

3.6 Translucent Sums

Translucent sums are weak sums with the dot elimination form that have been enriched in the following ways:

- The ability to give equational information about type components has been added
- Type equality has been modified to take this equational information into account

- They have been generalized to:
  - have any number of components
  - have named components
  - allow dependencies on previous (value) components
- Subtyping has been modified to allow
  - forgetting equational information about type components
  - reordering components
  - dropping components

The last two subtyping changes are to allow more flexible matching of module implementations to interfaces. This ability is very useful in practice because it allows modules to be combined in more flexible ways. In particular, it allows passing a module with extra or differently ordered components than a functor's argument interface requires to that functor without having to manually write a coercion function to extract only the required components. The reordering ability also frees programmers of the need to remember the exact order of an interface's components.

I shall discuss translucent sums in more detail in the next chapter.

3.7 First-Class Values

Like weak sums, on which translucent sums are based, translucent sums are ordinary values. This fact has a number of consequences. First, because modules are translucent sums, modules under my approach are first-class values. Second, because of this fact, modules may be nested inside other modules as value components (i.e., using the **val** keyword); my approach thus supports submodules without needing to handle submodule components specially.

Third, because having modules as first-class values means that any function can take or return modules, functors are simply ordinary (dependent) functions; accordingly, functors in my approach are also just ordinary values. Finally, fourth, because functions in λ-calculi are higher-order (they can take and return functions) and functors are just ordinary functions, my approach supports higher-order functors.

The recipe I have described builds a module system where modules are first-class values. It is possible to alter the recipe (see Section 13.8) to build instead a translucent higher-order module system where modules are second-class values. Such an approach would yield a system with the same abilities as the approach I am describing here except that it would not be possible to choose ADT implementations at runtime. The resulting system, however, would require substantially less complicated proofs.

Code using modules as first-class values may automatically lose type identity information in order to avoid unsoundness. Consider again, for example, the code to select a dictionary at runtime based on the number of items needed:

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

Suppose the programmer gave fully transparent interfaces to `LinkedList`, say `DICTIONARY` **with** `T=list(...)`, and `HashTable`, say `DICTIONARY` **with** `T = array(...)`. Because the type checker does not know what `n`'s value will be at runtime, it cannot safely determine `Dictionary.T`'s type identity. Accordingly, it must be conservative and assign `Dictionary` the more opaque interface `DICTIONARY`, which makes `Dictionary.T` abstract.

No special type rules are needed to implement this behavior: Because the type rule for if–then–else requires the then and else branches to have the same type and `LinkedList` and `HashTable` have different types, the type checker is forced to use subsumption to coerce both branches to a common supertype; all such types make `Dictionary.T` opaque. It is because of this ability to make types abstract when they cannot be determined at compile-time that my approach is able to treat modules as first-class values without risking unsoundness.

3.8 Linking

One advantage of a programming language with a higher-order module system based on type theory is that the process of separate compilation, including linking, can be described in the language without any need for additional constructs or type rules. The basic idea here is to describe separate compilation as a preprocessing stage that automatically converts files containing code with references to other modules into closed functors by using the files containing the interfaces for the referenced modules. The closed functors can then be compiled completely independently of any other files. The linker must then automatically generate code to apply the compiled functors generated by the compiler to each other in the correct order at runtime.

In order to give you a favor for how this process might work, I am now going to describe a very simple separate-compilation system. In this system, the implementation code for modules named M is placed in the file M`.m` while the interface for module M (only one interface per module is allowed in this system) is placed in the file M`.i`; this naming convention allows the preprocessor to find the interfaces of referenced modules.

Interface and implementation files in this system may optionally start with an import declaration. Only modules listed in a file's import declaration, if any, may be referenced

```
File contents of "String.i":
  interface
    type T;
    val  empty : T;
    ...
  end;

File contents of "String.m":
  module
    type T = array(char);
    val  empty = ...
    ...
  end;
```

Figure 3.1: Code for the `String` module

from that file. If a module's interface file includes an import declaration, then it's implementation file must include an import declaration which starts with the same list of modules imported in the module's interface file.

For an example program, consider Figures 3.1 and 3.2. Figure 3.1 contains the interface and implementation files for a module `String` that implements a string type from character arrays. Figure 3.2 contains the interface and implementation files for a module `Hash`, which implements an integer hash-table ADT keyed on strings using an associative list ADT. The modules `String` and `Assoc` (not shown) have no import declarations because they do not need to refer to any other modules. The `Hash` interface imports `String` (in order to use the type `String.T`), while the `Hash` implementation imports `Assoc` as well (in order to use the `String` and `Assoc` operations to implement hash tables).

The transform of `Hash` into a closed functor results in the code of Figure 3.3. This transform can be performed completely mechanically: the list of functor arguments comes from the implementation's import list, the interfaces from the appropriate interface files, and the functor body from the module's implementation file. Note that no references to external modules remain: The references to `String.T`, `Assoc.T`, and so on, refer to the `MkHash` functor's arguments, not to the external modules of the same name. Also note that checking that `Hash`'s implementation satisfies its publicly declared interface will be done automatically when the transformed code is compiled; the matching relation here is just subtyping on interfaces, avoiding the need for a separate set of rules. The transforms of `String` and `Assoc` (not shown) are much simpler since they import nothing.

When this program is linked together, the linker automatically generates the series

```
File contents of "Hash.i":
  import String;

  interface
    type T;
    val  new : unit -> T;
    val  lookup : T * String.T -> int;
    ...
  end;

File contents of "Hash.m":
  import String, Assoc;

  module
    type T = Assoc.T(int)
    fun  new() = ...
    fun  lookup(table:T, key:String.T) = ...;
    ...
  end;
```

Figure 3.2: Code for the `Hash` module

```
MkHash =
  functor (String: interface
                     type T;
                     val  empty : T;
                     ...
                   end,
           Assoc:  interface ... end)
             : interface
                 type T;
                 val  new : unit -> T;
                 val  lookup : T * String.T -> int;
                 ...
               end
     module
        type T = Assoc.T(int);
        fun  new() = ...
        fun  lookup(table:T, key:String.T) = ...;
        ...
     end;
```

Figure 3.3: Code for the transformed `Hash` module

```
String = MkString();

Assoc  = MkAssoc();

Hash   = MkHash(String, Assoc);
```

Figure 3.4: Code to combine the closed functors

of functor applications shown in Figure 3.4 to combine the closed functors together into a complete program. Determining what order to link the closed functors in is somewhat tricky because of dependencies between modules. In the simple system I am describing, the order is determined by first building a directed graph, whose nodes are the modules and whose edges represent dependencies between modules: An arrow is drawn from M to N iff M imports N or M appears before N in any import list. (The second part gives the programmer some control over the order of module initialization; this control can be important when side effects are involved.) Second, the graph is checked for cycles; it is an error, aborting linking, if any cycles are found. Otherwise, third, a topological sort is done to find a ordering that respects all the dependencies.

At runtime, this series of applications is done as the effect of the program. In a more realistic example, there would be a `Main` module, executed last, whose application would perform the desired computation. Note that if functors are used in the original program, then the transformed code will involve higher-order functors.

3.9 Type-Sharing Specifications

Standard ML has a feature that is sometimes useful called *type sharing*, which is used in interfaces to require that (sub)-component types are equal. To see an example of where this feature might be useful, imagine that we had a number of different ordered-dictionary implementations that we wished to test to make sure they worked correctly. One easy test we could do is to write a functor that takes in two ordered dictionaries and performs random operations on the two dictionaries in parallel; if their answers differ, then at least one of them must be incorrect.

In order to make this functor work, we are going to need to require that the two ordered dictionaries we are passed have the same key and element types. (Otherwise, we will not be able to perform the same operations on both ordered dictionaries.) This requirement can be expressed in SML by using type-sharing specifications:

```
functor RandomTest(structure x:ORDERED_DICTIONARY;
                   structure y:ORDERED_DICTIONARY;
                   ...
                     sharing type x.Key.T = y.Key.T
                     sharing type x.elem  = y.elem
                  ) = ...
```

(SML syntax is slightly different from my notation.)

This SML code can be translated into my approach by using a series of with expressions to set the required-equal type components equal to each other:

```
RandomTest =
  functor (x:ORDERED_DICTIONARY,
           y:ORDERED_DICTIONARY
               with Key.T = x.Key.T
               with elem  = x.elem,
           ...)
  ...
```

This basic idea — picking the type in an specified equivalence class with maximal scope and then setting the other types in that class equal to it using with statements — can used to translate any type-sharing specification from SML into my approach.

3.10 Summary

Summarizing, under my approach the identity of type components start out fully visible; this visibility can later be obscured either automatically when modules are used in a first-class manner in order to ensure soundness, or manually when the programmer inserts a type coercion in order to produce abstraction. This controlled visibility permits data abstraction where desired and modules as first-class values without limiting the uses of higher-order features that depend on modules "containing" other types. My approach also supports separate compilation because modules depend only on interfaces.

Since my approach gives the programmer control over the degree of visibility, she can choose opacity or transparency as needed, getting the best of both worlds. Indeed, because translucency permits the programmer to mix opacity and transparency in the same module (e.g., the interface for `HashTable`), new uses of higher-order features become possible (e.g., the B-Tree example). My approach also provides a number of other benefits:

- Functors are first-class
- Higher-order functors are provided
- Type-sharing specifications can be encoded
- Modules may export type abbreviations
- Linking and separate compilation can be described in the resulting programming language
- The resulting system has a simple and uniform type theory

Chapter 4

Technical Machinery

Now that I've described the basic ideas of my approach, I shall introduce in this chapter the technical machinery behind my approach. In particular, I shall explain the syntax and rules for translucent sums in some detail, describe how the various rules interact to obtain desirable properties such as decidable type equality, and motivate some restrictions in the rules. This chapter is intended to serve as an introduction to the machinery worked out in complete detail in the next part of this dissertation; accordingly, I ignore here a number of uninteresting minor technical issues such as the need for side conditions on many rules in order to avoid variable capture and duplicate field names.

4.1 Dot Notation

Traditionally, in order to make use of the contents of a weak-sum value (also called an existential value), a programmer must first *open* it. If the term M_1 evaluates to a weak sum of type **interface type** `T`; **val** `v`:A; **end**, then it may be opened for use in M_2 by writing **open** M_1 **as** `U,x` **in** M_2 **end**. During M_2's execution, `U` will be bound to the type component of the weak sum and `x` will be bound to its value component, which will have type [`U`/`T`]A. The type rule for `open` requires that M_2's type does not refer to `U`; this requirement is necessary to avoid unsoundness. (Because weak sums are first-class values, `U` may be bound to different types at runtime, making it unsafe to refer to it by a single static name outside the scope of the open statement.)

Having to write out an open statement each time you want to use a weak sum can be inconvenient; in a 1990 paper [11], Luca Cardelli and Xavier Leroy introduced a shortcut they called the dot notation. The basic idea is to allow the programmer to directly write `x.T` for the type component of and `x.v` for the value component of the weak sum denoted by the variable `x`. These direct references are then translated away by placing open statements next to the bindings of the weak sums and by substituting the names of the opened components for the direct references.

For example, consider the following piece of code:

```
let x = M₁ in
  x.v.1(λs::x.T. true)
end
```

This code translates into the following piece of code:

```
let x = M₁ in
  open x as x_T,x_v in
    x_v.1(λs:x_T. true)
  end
end
```

By using a more complicated translation, Cardelli and Leroy showed, among other things, that their dot notation could be extended from working on just variables to working on any (sub)-component of a variable as well. Under that extension, if `x.y.z`, the `y.z` sub-component of the variable `x`, is a weak sum, then its type component could be referred to directly as `x.y.z.T`. The extension does not compromise soundness because the (sub)-components of variables are essentially just another kind of variable.

I have chosen to build the dot notation into translucent sums directly instead of requiring it to be first translated away or not providing it. In their paper, Cardelli and Leroy do not deal with dependent sums or functions; it seems unlikely that the dot notation can be translated into open statements in the presence of these features, both of which are present in my system. Thus, the dot notation in my system should be regarded as an independent feature and not an abbreviation for the use of open. As I shall explain in Section 4.5, my approach is able to support the dot notation on a somewhat larger class of terms than variables and their (sub)-components.

4.2 Creation

For the this chapter, I shall use the following syntax:

Simple Modules	S	::=	**module** B_1 ; $\cdots$ B_n ; **end**	$(n \geq 0)$
Interfaces	I	::=	**interface** D_1 ; $\cdots$ D_n ; **end**	$(n \geq 0)$
Bindings	B	::=	**val** x=M | **type** α=A	
Declarations	D	::=	**val** x:A | **type** α=A | **type** α	
Terms	M	::=	x | S | $M.x$ | **functor** $(x{:}A)$ M | $M_1\,M_2$ | ...	
Types	A	::=	α | I | **FUNCTOR** $(x{:}A){:}A'$ | $M.\alpha$ | ...	
Assignments	Γ	::=	$D_1;\ \cdots\ D_n$	$(n \geq 0)$

Here, the metavariable α ranges over *type variables* and the metavariable x ranges over *term variables*. I have omitted the syntax for constructs other than translucent sums and dependent functions because it will not be needed; likewise, I have simplified the presentation of translucent sums here by using only one name per field rather than the pair of names — one for internal access and one for external access — that are required for technical reasons (see Section 13.1). In the next chapter I shall introduce a more concise, precise, and formal notation for my system.

In order to type check a simple module, we type check each of its bindings under the preceding bindings, if any, and the current assignment (assignments describe the variables from the rest of the program currently in scope):

$$\frac{\forall i \in [1..n].\ \Gamma; D_1; \cdots; D_{i-1} \vdash B_i : D_i}{\Gamma \vdash \textbf{module}\ B_1; \cdots; B_n;\ \textbf{end} : \textbf{interface}\ D_1; \cdots; D_n;\ \textbf{end}} \qquad \text{(I-SIMPLE)}$$

The resulting type for the module is an interface type listing the "types" of the module's bindings. The "type" of a binding is a declaration; the following rules associate declarations with bindings:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \textbf{val}\ x{=}M : \textbf{val}\ x{:}A} \qquad \text{(I-VAL)}$$

$$\frac{\Gamma \vdash A \text{ valid}}{\Gamma \vdash \textbf{type}\ \alpha{=}A : \textbf{type}\ \alpha{=}A} \qquad \text{(I-TYPE)}$$

Here $\Gamma \vdash M : A$ denotes that term M has type A under assignment Γ and $\Gamma \vdash A$ valid denotes that A is a valid type under Γ. Note that the rules assign transparent declarations (**type** α=A) rather than opaque declarations (**type** α) to type bindings. This behavior implements the default interface policy for simple modules (full transparency) that I described in Section 3.2.

As an example, these rules assign **module type** α`=int;` **val** `a=2;` **val** `b=a+2;` **end** the type **interface type** α`=int;` **val** `a:int;` **val** `b:int;` **end** under Γ. Note that because we type check successive bindings in the scope of the previous ones, fields can refer to earlier fields. In the above example, the `b` field was able to refer to the `a` field because it was type checked under the assignment Γ;**type** α=int;**val** x:int. Earlier type fields such as α may also be referred to in later fields.

Later fields may also refer to (sub)-components of earlier fields. For example, the following module is well typed:

```
module val  P=module
                 type T=int;
                 val  x=3;
               end;
            type U=P.T;
            val  y=P.x;
end
```

Note the use of the dot notation here to refer directly to the type `P.T`. The rule for determining when such a reference is valid is (roughly) as follows:

$$\frac{\Gamma \vdash M : \textbf{interface}\ \Gamma_1; \textbf{type}\ \alpha; \Gamma_2;\ \textbf{end}}{\Gamma \vdash M.\alpha \text{ valid}} \qquad \text{(V-DOT)}$$

That is to say, if M denotes a translucent sum with an α type field[1] under Γ, then $M.\alpha$ denotes a type. This rule is in fact too general; I shall explain why and how the set of terms that this rule applies to needs to be restricted in Section 4.5. I shall defer discussion of how $M.x$ is typed until Section 4.6.

The ability to refer to earlier fields also exists in interfaces:

$$\frac{\forall i \in [1..n].\ \Gamma; D_1; \cdots; D_{i-1} \vdash D_i \text{ valid}}{\Gamma \vdash \textbf{interface}\ D_1; \cdots; D_n;\ \textbf{end} \text{ valid}} \qquad \text{(V-INTER)}$$

Thus, interfaces like **interface type** α; **val** `x`:α; **end** are valid.

All of the rules for dependent functions (functors) are standard except for the typing rule for application, which I shall explain in Section 4.6. For example, the rule for type

[1] Note that a transparent declaration for α in M's interface can always be turned into an opaque declaration for α by using subsumption on M to forget the identity of α.

checking functors is as follows:

$$\frac{\Gamma; x{:}A \vdash M : A'}{\Gamma \vdash \textbf{functor}\ \ (x{:}A)\ \ M : \textbf{FUNCTOR}\ \ (x{:}A){:}A'} \qquad \text{(I-FUNC)}$$

Note that the result type of a functor (A') can depend on its argument (x).

4.3 Equality

Type equality in my system is structural (i.e.., component-wise) except for equations derived from the information about type identities found in translucent sums and from reductions involving functions on types (not discussed in the first part of this dissertation; see Section 5.1). In particular, two interfaces are equal if their corresponding declarations are equal:

$$\frac{\forall i \in [1..n].\ \Gamma; D_1; \cdots; D_{i-1} \vdash D_i = D_i'}{\Gamma \vdash \textbf{interface}\ D_1; \cdots; D_n;\ \textbf{end} = \textbf{interface}\ D_1'; \cdots; D_n';\ \textbf{end}} \qquad \text{(E-INTER)}$$

Two declarations are equal if they are of the same sort, declare the same variable, and contain equal types:

$$\frac{\Gamma \vdash A = A'}{\Gamma \vdash \textbf{val}\ x{:}A = \textbf{val}\ x{:}A'} \qquad \text{(E-VAL)}$$

$$\Gamma \vdash \textbf{type}\ \alpha = \textbf{type}\ \alpha \qquad \text{(E-TYPE-O)}$$

$$\frac{\Gamma \vdash A = A'}{\Gamma \vdash (\textbf{type}\ \alpha{=}A) = (\textbf{type}\ \alpha{=}A')} \qquad \text{(E-TYPE-T)}$$

Note that equality does not permit the reordering of translucent-sum fields.

The information about type identities contained in translucent sums leads to equations in two ways. First, whenever a transparent declaration occurs in an assignment, it gives rise to an equation involving its type variable:

$$\Gamma; \textbf{type}\ \alpha{=}A; \Gamma' \vdash \alpha = A \qquad \text{(ABBREV)}$$

Because the components of translucent sums (and functors) are compared for equality under an assignment extended with the previous declarations, if any, this rule give rise to equations like the following:

$$\begin{array}{l} \textbf{interface type}\ \alpha\texttt{=int};\ \textbf{val}\ \texttt{x}{:}\alpha;\quad \textbf{end}\quad = \\ \textbf{interface type}\ \alpha\texttt{=int};\ \textbf{val}\ \texttt{x:int};\ \textbf{end} \end{array}$$

Second, declarations of translucent sums containing transparent declarations, either directly or in a sub-component, also give rise to equations:

$$\frac{\mathrm{FV}(A) \cap \mathrm{fields}(\Gamma_1) = \emptyset \qquad \Gamma \vdash M : \textbf{interface}\ \Gamma_1; \textbf{type}\ \alpha{=}A; \Gamma_2;\ \textbf{end}}{\Gamma \vdash M.\alpha = A} \qquad (\mathrm{ABBREV}')$$

As with the previous rule involving the dot notation, V-DOT, this rule is too general; a similar restriction to the set of terms it applies to will need to be made.

I use $\mathrm{FV}(A)$ to denote the free variables of A and $\mathrm{fields}(\Gamma)$ to denote the fields declared in Γ. Thus, the first precondition of the ABBREV′ rule expresses the requirement that the type α is equal to (A) does not *depend* on any earlier fields in M's type; this precondition is needed to prevent variable capture.

Such dependencies can be classified as either *essential* or *inessential*. A dependency is inessential if it can be removed by the use of the equality rules alone; otherwise, it is essential. Dependencies on transparently-defined fields are inessential: they can be removed by using the ABBREV or the ABBREV′ rule as appropriate to replace the field name with its declared identity. (The typing rules prohibit fields from depending on themselves.) Dependencies on opaquely defined fields, on the other hand, are usually essential.

As an example, in the following interface type the `z` field depends essentially on the α field but only inessentially on the β field:

```
interface
  type α;
  type β = int;
  val  z: FUNCTOR (s:α):β;
end
```

The difference between essential and inessential dependencies is important because several type rules in my system require all dependencies between certain fields to be removed before they can be applied.

Like the ABBREV rule, the ABBREV′ rule also gives rise to equations on interfaces when combined with the component-wise equality rules. For example, we have that

```
interface
  val  x: interface type T=int; end;
  type U = x.T;
end
```

is equal to

```
interface
  val  x: interface type T=int; end;
  type U = int;
end
```

4.4 Subtyping

Subtyping in my system is structural modulo equality — equal types are always subtypes of each other — except for forgetting information about type-component identities and the reordering and dropping of translucent-sum fields. The component-wise subtyping rule for translucent sums says that two interfaces are subtypes if their corresponding declarations are "subtypes":

$$\frac{\forall i \in [1..n].\ \Gamma; D_1; \cdots; D_{i-1} \vdash D_i \leq D'_i \qquad \Gamma \vdash \textbf{interface}\ D'_1; \cdots; D'_n;\ \textbf{end}\ \text{valid}}{\Gamma \vdash \textbf{interface}\ D_1; \cdots; D_n;\ \textbf{end} \leq \textbf{interface}\ D'_1; \cdots; D'_n;\ \textbf{end}} \qquad \text{(S-INTER)}$$

One declaration is a structural "subtype" of another if it is of the same sort and declares the same variable either to be a subtype of the other declaration's declared type (value-declaration case) or to be equal to the same type(s) modulo equality (type-declaration case):

$$\frac{\Gamma \vdash A \leq A'}{\Gamma \vdash \textbf{val}\ x{:}A \leq \textbf{val}\ x{:}A'} \qquad \text{(S-VAL)}$$

$$\Gamma \vdash \textbf{type}\ \alpha \leq \textbf{type}\ \alpha \qquad \text{(S-TYPE-O)}$$

$$\frac{\Gamma \vdash A = A'}{\Gamma \vdash (\textbf{type}\ \alpha{=}A) \leq (\textbf{type}\ \alpha{=}A')} \qquad \text{(S-TYPE-T)}$$

Note that it is not safe to replace a transparent type declaration with one that declares the field name to be equal to a subtype of the original type because type fields can be used in contravariant positions; types in such positions can in general only be replaced safely with supertypes.

The forgetting of type-identity information is implemented by a non-structural rule for declaration "subtyping":

$$\frac{\Gamma \vdash A\ \text{valid}}{\Gamma \vdash (\textbf{type}\ \alpha{=}A) \leq (\textbf{type}\ \alpha)} \qquad \text{(S-TYPE-F)}$$

Some examples of subtyping using these rules are as follows:

```
(a)      interface type T=int; val x:int; end
       ≤ interface type T=int; val x:T;   end
       ≤ interface type T;     val x:T;   end

(b)      interface type T; type U=T; end
       ≤ interface type T; type U;   end
```

Here in (a), we first created an inessential dependency of the `x` field on `T` using equality then converted it into an essential dependency by using forgetting to make `T` an opaque field. In (b), by contrast, we broke an essential dependency of the `U` field on `T` by forgetting that `U` was equal to `T`. Subtyping, thus, unlike equality, can be used to break or create essential dependencies.

Note that before a field's type-identity information can be forgotten, it may be necessary to remove some dependencies on that field in order to ensure that the resulting type is valid as required by the second precondition of rule S-INTER. (Such dependencies can always be removed by substituting the type the field is declared equal to for the field name everywhere.) Consider the following valid interface:

```
interface
  type S = interface type T; end;
  val  M:S;
  val  N:S;
  val  x:M.T;
end
```

We cannot directly forget `S`'s identity because the resulting type is invalid:

```
interface
  type S;
  val  M:S;
  val  N:S;
  val  x:M.T;
end
```

(Here, `M.T` is an invalid type because `M`, being of a (now) abstract type, cannot be shown to be a translucent sum.)

However, we can forget `S`'s identity if we first replace the occurrence of `S` in `M`'s type with the interface it is currently defined equal to, resulting in the following type:

```
interface
  type S;
  val M : interface type T; end;
  val N:S;
  val x:M.T;
end
```

In addition to the component-wise subtyping rule for interfaces, there are additional non-structural rules that permit the reordering and dropping of translucent-sum fields. Because it is unclear how the rules that implement this behavior should be formulated (see Section 13.2), I shall just give a very rough description of their behavior here: Fields in interfaces may be dropped if no other fields (essentially) depend on them. A field may be moved to the right past other fields if they do not depend (essentially) on the moving field. Some variants of these rules allow a field to be moved to the right past fields that depend essentially on the moving field under some conditions. See Section 13.2 for details.

Consider, for example, the following interface:

```
interface type a; type b=int; val c:b; val d:a; end
```

Here, the `c` and `d` fields could be dropped directly and the `b` field could be dropped after the dependency on it by the `c` field had been broken by using equality. The `a` field, by contrast, cannot be dropped unless the `d` field is dropped first because the `d` field depends on it essentially. The `a` field could be moved right past the `b` and `c` fields but not past the `d` field. Likewise, the `b` field could be moved right past the `c` and `d` fields only if the dependency by `c` on `b` was first broken.

Note that, like the rule for ordinary non-dependent functions, the subtyping rule for functor types is contravariant:

$$\frac{\Gamma \vdash A_2 \leq A_1 \qquad \Gamma; x{:}A_2 \vdash A_1' \leq A_2'}{\Gamma \vdash \textbf{FUNCTOR}\ (x : A_1) : A_1' \leq \textbf{FUNCTOR}\ (x : A_2) : A_2'} \qquad \text{(S-FUNC)}$$

Here, the direction of subtyping on the functor argument types (A_i) is reversed from that on the overall types. This fact means that subtyping can be used to "remember" the type-identity information of variables declared in contravariant positions. This ability provides another way for dependencies to be broken:

```
   FUNCTOR (x:interface type T;     end):x.T
≤  FUNCTOR (x:interface type T=int; end):x.T
≤  FUNCTOR (x:interface type T=int; end):int
```

Here, remembering followed by equality was used to break the essential dependency of the result type on the argument `x`.

4.5 The Phase Distinction

One desirable property of a programming language is that it have a *phase distinction* [22]. A programming language is said to have a phase distinction if programs in it can be type checked without evaluating general program expressions (terms). Programming languages without a phase distinction are hard to compile efficiently; in the presence of side effects, their semantics are also unclear. (What does it mean for a program's type to be "if today is Monday then `int` else `bool`"?)

Surprisingly, my system has a phase distinction in spite of allowing modules as first-class values. The only difficult part to ensuring a phase distinction for my system is figuring out how to handle equality on types containing terms. In particular, how should equations like the following be checked?

$$M.\alpha = N.\alpha$$

The naive method is to simply evaluate M and N, read off the types of their α type components from the resulting values, and then compare the types using equality. However, since M and N may be arbitrary terms, this rules out having a phase distinction.

My answer is to restrict the set of terms that may occur in types to an extended set of "values" I call *extended values*. The set of extended values consists of the usual call-by-value values (denoted by the metavariable V) extended with term variables and selections on extended values:

Definition 4.5.1 (Syntax)
Extended Values ν $::=$ $x \mid V \mid \nu.y$

I choose this set because it allows referring to any previous type component (including nested cases like `x.y.`α) and is closed under substitution of a call-by-value value for a term variable, making defining the call-by-value semantics of function application much easier. Note that type coercions of the form $V{<:}A$ are not considered call-by-value values because they can be reduced further to V. If they were added to the set of extended values, then the identity of types like (**module type** `T=int;` **end** `<:` **interface type** `T;` **end**)`.T` becomes problematic because in order for abstraction to work properly, this type should not even be equal to itself, let alone be equal to `int`.

This restriction, which I call the extended-value restriction, is implemented both syntactically, by replacing $M.\alpha$ in the grammar for types with $\nu.\alpha$, and via the type rules, by replacing M by ν in the rules V-DOT and ABBREV$'$. The restriction makes computing the identity of a type component like $\nu.\alpha$ easy: The only operation that needs to be performed is selection, which never produces side effects and cannot cause non-termination.

The computing of type identities for $\nu.\alpha$'s is done in my system by the ABBREV$'$ rule combined with the rules for typing simple modules and selections. These rules give rise to equations like the following:

```
(module type T=int; val x=3; end).T = int
(module type T=int; type U=T*bool; end).U = int*bool
(module val x = module type T=int; end; end.x).T = int
```

Using these equations, any valid type in my system can be converted using equality into an equal type that contains terms of only the form $x.y_1.y_2\cdots.y_n$, $n \geq 0$. I shall call such terms *places*. (The valid-type requirement here is to rule out cases with invalid selections like $(\lambda x{:}\texttt{int}.\,3).\alpha$.) Once the need for evaluating terms has been removed, it is easy to establish a phase distinction.

Note that the remaining terms are of the form that Cardelli and Leroy showed how to convert into open statements (see Section 4.1); this fact indicates that my extension of the dot notation to extended values is conservative because $\nu.\alpha$'s can be regarded as just syntactic sugar for a type using only the unextended dot notation.

One consequence of the extended-value restriction is that my system is limited to call-by-value: Call-by-name requires the ability to substitute general terms for term variables in terms, which in turn requires the ability to substitute general terms for term variables in types because terms may contain types; however, substituting general terms into types results in types containing general terms in violation of the restriction.

Attempting to relax the extended-value restriction in order to get around this limitation gives unsoundness in the presence of side effects. To see this, consider the following example, where the `alternate` function alternates between returning `true` and `false` (alternate uses side effects internally):

```
IntPkg  = module type T = int;
                 val  v = 3;
                 val  f = negate;
          end;

BoolPkg = module type T = bool;
                 val  v = true;
                 val  f = not;
          end;
```

```
Apply   = functor (Pkg: interface type T;
                                  val  v:T;
                                  val  f:FUNCTOR (x:T):T;
                        end)
            begin
              Pkg.f(Pkg.v);
              true;
            end;

Apply(if alternate() then IntPkg else BoolPkg)
```

This code is well typed in my system[2] and safe under call-by-value because `Apply` under call-by-value applies the function `f` to the value v from the same package. However, if we evaluate this code under call-by-name, we evaluate `Apply`'s argument twice resulting in a type error because we end up evaluating either `IntPkg.f(BoolPkg.v) = negate(true)` or `BoolPkg.f(IntPkg.v) = not(3)`. A similar runtime type error occurs if we first β-reduce or inline the call to `Apply`, then evaluate using call-by-value. Because the β-expansion would type check if the extended-value restriction were relaxed, this example shows that such a relaxation is unsound in the presence of effects.

4.6 Selection and Application

Another consequence of the extended-value restriction is that the traditional elimination rules[3] for dependent sums and functions cannot be used because they require the ability to substitute general terms into types:

$$\frac{\Gamma \vdash M : \textbf{interface val } \mathtt{x}{:}A_1;\ \textbf{val } \mathtt{y}{:}A_2;\ \textbf{end}}{\Gamma \vdash M.\mathtt{x} : A_1} \qquad \text{(I-TRAD-S1)}$$

$$\frac{\Gamma \vdash M : \textbf{interface val } \mathtt{x}{:}A_1;\ \textbf{val } \mathtt{y}{:}A_2;\ \textbf{end}}{\Gamma \vdash M.\mathtt{y} : [M.\mathtt{x}/\mathtt{x}]A_2} \qquad \text{(I-TRAD-S2)}$$

$$\frac{\Gamma \vdash M_1 : \textbf{FUNCTOR } (x{:}A){:}A' \qquad \Gamma \vdash M_2 : A}{\Gamma \vdash M_1\, M_2 : [M_2/x]A'} \qquad \text{(I-TRAD-F)}$$

[2] I have used an imperative version of `Apply` here, which executes `Pkg.f(Pkg.v)` solely for its side effects, in order to avoid a restriction on functor applications that I discuss in the next section.

[3] Elimination rules for a construct handle type checking that construct's elimination form(s), in this case selection for dependent sums and application for dependent functions.

Here, I have simplified things by only giving the case for binary dependent sums and have used $[M/x]A$ to denote the substitution of the term M for the term variable x in the type A in a variable capture avoiding way.

In order to avoid this problem, I limit my selection and application rules to the non-dependent cases, which do not require term substitution to handle:

$$\frac{\Gamma \vdash \nu : \textbf{interface } \Gamma_1; \textbf{type } \alpha; \Gamma_2; \ \textbf{end}}{\Gamma \vdash \nu.\alpha \text{ valid}} \tag{V-DOT}$$

$$\frac{\begin{array}{c}\mathrm{FV}(A) \cap \mathrm{fields}(\Gamma_1) = \emptyset \\ \Gamma \vdash \nu : \textbf{interface } \Gamma_1; \textbf{type } \alpha{=}A; \Gamma_2; \ \textbf{end}\end{array}}{\Gamma \vdash \nu.\alpha = A} \tag{ABBREV$'$}$$

$$\frac{\begin{array}{c}\mathrm{FV}(A) \cap \mathrm{fields}(\Gamma_1) = \emptyset \\ \Gamma \vdash M : \textbf{interface } \Gamma_1; \textbf{val } x{:}A; \Gamma_2; \ \textbf{end}\end{array}}{\Gamma \vdash M.x : A} \tag{I-SELECT}$$

$$\frac{\begin{array}{l}\Gamma \vdash M_2 : A \\ \Gamma \vdash M_1 : \textbf{FUNCTOR } \ (x{:}A){:}A' \qquad x \notin \mathrm{FV}(A')\end{array}}{\Gamma \vdash M_1\, M_2 : A'} \tag{I-APP}$$

When can these rules be applied if we assume that all of their preconditions except the no-dependency one(s) are met? Clearly, the V-DOT rule can always be applied because opaque declarations cannot depend on anything. Less obviously, if M can be given a *fully-defined transparent type* — a type that declares all of M's type (sub)-components transparently — that also satisfies the traditional preconditions then the I-SELECT rule can be applied to M and, furthermore, if M is an extended value, then the ABBREV$'$ rule can be applied to it as well. This fact holds because all dependencies within and on a fully-defined transparent type must be inessential and hence breakable via subsumption; moreover, this breaking can be done without invalidating the traditional preconditions.

The case for the I-APP rule is similar but the reasoning is more complicated. Here, it is the argument M_2 that needs to be given a fully-defined transparent type in order for the rule to apply. For the traditional preconditions to hold, this type must be a subtype of M_1's domain type. Given these two conditions, the I-APP rule can always be applied: first use subtyping to specialize the functor M_1's domain type to the fully-defined transparent type for M_2; second, use equality to break all dependencies by the functor result type (A') on the functor argument (x), which will now all be inessential; and third, apply the I-APP rule to the resulting non-dependent function type.

As an example, consider type checking the application of a functional version of `Apply` (`FApply`, defined below) to `IntPkg` (defined in the previous section).

```
FApply  = functor (Pkg: interface type T;
                                   val   v:T;
                                   val   f:FUNCTOR (x:T):T;
                             end)
              Pkg.f(Pkg.v);
```

FApply and IntPkg have the following types:

```
IntPkg : interface type T=int;
                  val v:int;
                  val f:FUNCTOR (x:int):int;
         end

FApply : FUNCTOR (Pkg: interface
                         type T;
                         val   v:T;
                         val   f:FUNCTOR (x:T):T;
                       end):Pkg.T
```

Note that IntPkg's type is a fully-defined transparent subtype of FApply's domain type (use equality to replace occurrences of int with T then forget the type identity of T) so the functor application will be well typed if we can remove the dependency on Pkg by FApply's result type.

By using subtyping via subsumption, we can specialize FApply's domain type to IntPkg's type:

```
FApply : FUNCTOR (Pkg: interface
                         type T=int;
                         val   v:int;
                         val   f:FUNCTOR (x:int):int;
                       end):Pkg.T
```

Next, we can use equality via subsumption to replace all occurrences of Pkg.T in FApply's result type by int; this step breaks the dependency on FApply's domain type by its result type:

```
FApply : FUNCTOR (Pkg: interface
                         type T=int;
                         val   v:int;
                         val   f:FUNCTOR (x:int):int;
                       end):int
```

Finally, we can apply the resulting non-dependent version of `FApply` to `IntPkg` using I-APP to get that `FApply(IntPkg)` has type `int`. Note that we cannot type `FApply(IntPkg)` by first using subsumption to coerce M_2's type to that of `FApply`'s domain type and then using I-APP because once the information about `IntPkg.T` is lost there is no way to break the dependency on `Pkg.T` by `FApply`'s result type.

Because any term that does not denote a translucent sum will have a fully-defined transparent type (because it will have no type (sub)-components), the above result means that the application of a functor to any such term will never fail due to the no-dependency precondition. Thus, functors act like standard dependent functions when applied to anything other than a translucent sum.

There are two special cases worthy of note where translucent-sum expressions can always be given fully-defined transparent types. The first such case is translucent sums that contain no type (sub)-components. Such translucent sums are essentially ordinary records; their types are always fully-defined transparent types with no dependencies between fields because they contain no type (sub)-components, which could be depended on.

The second case concerns translucent-sum extended values; in the next section I shall introduce two rules I call the EVALUE rules, which allow a fully-defined transparent type to be given to any extended value without losing any information about its fields. This ability means that $\nu.\alpha$, $\nu.x$, and $F(\nu)$ will always type check if ν has an α type field, ν has an x type field, or ν is a member of F's domain type, respectively. Because the ABBREV′ rule is restricted to working on only extended values, this fact means that the no-dependency restriction does not in fact restrict the occasions when the ABBREV′ rule can be applied. Note also that module names (e.g., `IntPkg` or `SearchTree.Binary`) are special cases of extended values; hence, selection of and application to a named module cannot fail due to the no-dependency restriction.

What about the remaining cases where the no-dependency restriction cannot be met? By the process of elimination, these are cases where we are computing a new module (rather than referring to a previously computed module by name), which contains at least one type component that is depended on by the resulting type and whose identity the type checker does not know. Recall that unknown type components can only occur when the programmer either uses modules in a first-class manner or uses a type coercion to intentionally forget the identity of a type component.

These remaining cases need to be blocked in order to avoid unsoundness. The intuition behind this fact is that in these cases the result type of the expression in question contains a type that we have no name for; because we have no name for that type, we cannot be sure that it does not vary at runtime, leading to possible unsoundness. (Because my system uses call-by-value, type names must denote a constant type in each of their dynamic scopes.) As an example, consider the following unsafe code:

```
(if alternate() then IntPkg else BoolPkg).f(
  (if alternate() then IntPkg else BoolPkg).v)
```

This is the β-reduction of the unsoundness example from Section 4.5. Note that it would be well typed if we removed the no-dependency restriction on selection and used the traditional selection rules for dependent sums. This example can be η-expanded to produce an similar unsoundness example for functor application:

```
SelectF = functor (Pkg: interface type T;
                                  val  v:T;
                                  val  f:FUNCTOR (x:T):T;
                        end)
            Pkg.f;

SelectV = functor (Pkg: interface type T;
                                  val  v:T;
                                  val  f:FUNCTOR (x:T):T;
                        end)
            Pkg.v;

SelectF(if alternate() then IntPkg else BoolPkg)(
  SelectV(if alternate() then IntPkg else BoolPkg))
```

Note that this example does not violate the no-dependency restriction on selection because selection is applied only to module names and that both `SelectF` and `SelectV` are well typed even in the presence of the no-dependency restriction.

4.7 The EVALUE rules

Suppose the current assignment contains the following declaration:

val `P` : **interface type** `T;` **val** `x:T;` **end**

What types can we give to the expression `P` under this assignment? Because we have a name, `P`, for the translucent-sum expression we are trying to type, we have a name for the contents of its `T` field, namely `P.T`. This suggests that we can give `P` the type **interface type** `T=P.T;` **val** `x:P.T;` **end**, which is a subtype of the declared type for `P`.

This technique of giving a more expressive type to translucent sums when we have a name for their type components can be generalized to work on arbitrary translucent-sum extended values. The name in this case is simply ν`.T` where ν is the extended value in question. The technique cannot be extended to general terms because of the extended-value restriction (M.T is not in general a valid type in my system). The following two typing rules implement this technique:

$$\frac{\Gamma \vdash \nu : \textbf{interface } \Gamma_1;\ \textbf{type } \alpha;\ \Gamma_2;\ \textbf{end}}{\Gamma \vdash \nu : \textbf{interface } \Gamma_1;\ \textbf{type } \alpha\texttt{=}\nu\texttt{.}\alpha;\ \Gamma_2;\ \textbf{end}} \qquad \text{(EVALUE-T)}$$

$$\frac{\Gamma \vdash \nu.x : A' \qquad \Gamma \vdash \nu : \textbf{interface } \Gamma_1;\ \textbf{val } x\texttt{:}A;\ \Gamma_2;\ \textbf{end}}{\Gamma \vdash \nu : \textbf{interface } \Gamma_1;\ \textbf{val } x\texttt{'}\texttt{:}A;\ \Gamma_2;\ \textbf{end}} \qquad \text{(EVALUE-V)}$$

(The EVALUE-V rule is used in cases of nested translucent sums to apply the technique recursively.)

We can give any extended value ν a transparent type using these rules by repeating the following sequence of steps as many times as possible: first, find the leftmost type (sub)-component in ν's current type that is not declared transparently, call it C. Second, use equality (via subsumption) to break all dependencies on earlier type (sub)-components in ν's type. (We can do this breaking because all earlier type (sub)-components must be declared transparently by step one.) Third, use the EVALUE rules to convert the C field from an opaque declaration to a transparent one. (Step two here is necessary to ensure that the selection in the EVALUE-V rule does not fail due to the no-dependency restriction.) Note that equality (via subsumption) may be needed before applying the EVALUE rules in order to rewrite ν's type into the form **interface** ... **end**.

If ν's original type was *fully-defined* (it declares all its type (sub)-components), then the resulting type will be a fully-defined transparent type. A fully-defined type can be found for ν by simply type checking ν without reordering or dropping fields from ν and its (sub)-components. This procedure can always be done if ν is well typed (adding extra information does not make well-typed terms ill-typed). Thus, we can give a fully-defined transparent type to any well-typed extended value using the EVALUE rules.

As an example, consider giving a fully-defined transparent type to the extended value `x` under an assignment that contains the following declaration:

val `x`:**interface type** `T`; **val** `y`:**interface type** `U`; **type** `V=T`; **end**; **end**

The extended value `x` goes through the following types as we give it a transparent type:

interface type `T`; **val** `y`:**interface type** `U`; **type** `V=T`; **end**; **end**
interface type `T=x.T`; **val** `y`:**interface type** `U`; **type** `V=T`; **end**; **end**
interface type `T=x.T`; **val** `y`:**interface type** `U`; **type** `V=x.T`; **end**; **end**
interface type `T=x.T`; **val** `y`:**interface type** `U=x.y.U`;
type `V=x.T`; **end**; **end**

In addition to their interactions with the elimination rules, the EVALUE rules are important because they prevent the identity of opaquely-defined type (sub)-components from being lost when the (sub)-components are given a new name. Without these rules, an abstract type and its copy would be considered different abstract types. Consider, for example, the following code fragment:

```
M =    module   type T=int; type U=int; type V=bool; end
    <: interface type T;     type U=T;   type V=bool; end;

N = M;

O = module val X=N; end;
```

Using the EVALUE rules we can give the following transparent types to these module names:

```
M : interface type T=M.T; type U=M.T; type V=bool; end

N : interface type T=M.T; type U=M.T; type V=bool; end

O : interface
      val X : interface type T = M.T;
                        type U = M.T;
                        type V = bool;
              end;
    end
```

Accordingly, we can infer that `M.T = M.U = N.T = N.U = O.X.T = O.X.U` and that `M.V = N.V = O.X.V = bool`, but not that `M.T = int`. These equations mean that `M`, `N`, and `O.X` are interchangeable. Without the EVALUE rules we would only be able to infer that `M.T = M.U`, `N.T = N.U`, `O.X.T = O.X.U`, and `M.V = N.V = O.X.V = bool`; `M`, `N`, and `O.X` are not interchangeable in that case because they do not contain equivalent `T` components.

Note that the EVALUE rules, the elimination rules for selection and application, and the ABBREV rules working in concert are able to propagate all the needed information about the identity of type (sub)-components solely through types; no term (even just a value) ever needs to be substituted during type checking. This machinery is what allows my system to support separate compilation: All the information necessary for type checking can be placed in a module's type (interface); there is no need to have the code of a supplier module so that it may be substituted.

One way to show the power of this machinery is to show that all the information available from substituting an extended value ν for x can also be made available by giving x an appropriate type A possessed by ν (i.e., a fully-defined transparent type as discussed earlier). This claim can be formalized as the following proposition: If ν is a well-typed extended value under Γ then there exists a type A such that A is a *minimal* type for ν under Γ and for all types A' such that $\Gamma; [\nu/x]\Gamma' \vdash [\nu/x]A'$ valid, we have that $\Gamma; x{:}A; \Gamma' \vdash [\nu/x]A' = A'$. The type A is a transparent minimal type for M under Γ if for all types A' such that $\Gamma \vdash M : A'$, we have that $\Gamma \vdash A \leq A'$. I prove a version of this proposition in Sections 11.5 and 11.6 (see especially Theorem 11.5.10 and Lemma 11.6.10) for my kernel system.

This proposition can be used to show that the following versions of the traditional elimination rules, which have been restricted to extended values, can be derived in my system:

$$\frac{\Gamma \vdash \nu : \textbf{interface val } \mathtt{x}{:}A_1\,;\ \textbf{val } \mathtt{y}{:}A_2\,;\ \textbf{end}}{\Gamma \vdash \nu.\mathtt{y} : [\nu.\mathtt{x}/\mathtt{x}]A_2} \qquad \text{(I-TRAD-S2')}$$

$$\frac{\Gamma \vdash M : \textbf{FUNCTOR } (x{:}A){:}A' \qquad \Gamma \vdash \nu : A}{\Gamma \vdash M\,\nu : [\nu/x]A'} \qquad \text{(I-TRAD-F')}$$

(For simplicity, I again show only the binary sum case here.)

I shall just sketch the proof of the first rule here: Let ν have the type **interface val** $\mathtt{x}{:}A_1$; **val** $\mathtt{y}{:}A_2$; **end** under Γ. By applying the proposition, we can get a transparent minimal type for $\nu.\mathtt{x}$, call it A_1'. By EVALUE-V then, ν must also have the type **interface val** $\mathtt{x}{:}A_1'$; **val** $\mathtt{y}{:}A_2$; **end** under Γ. By the second part of the proposition then, this type is equal to **interface val** $\mathtt{x}{:}A_1'$; **val** $\mathtt{y}{:}[\nu.\mathtt{x}/\mathtt{x}]A_2$; **end**. Note that the `y` field of this type cannot depend on the `x` field because it contains no `x`'s (ν cannot contain `x` because of (unshown) side conditions required to prevent variable capture). Accordingly, we can apply equality (via subsumption) to give ν this type, then I-SELECT to show that $\nu.y$ has the desired type $[\nu.\mathtt{x}/\mathtt{x}]A_2$.

Thus, for selection from and application to extended values, my elimination rules are equivalent to the traditional ones. A similar result can be shown for the other special cases (non–translucent-sum values and translucent sums without type (sub)-components) mentioned before in Section 4.6. My rules differ in the remaining cases in that they refuse to allow some operations that the traditional rules permit in order to prevent unsoundness in the presence of modules as first-class values and to avoid the need for substituting general terms so as to be able to support separate compilation.

Chapter 15

Conclusions

In this, the final chapter, I sum up the contributions of my dissertation and discuss possible future work.

15.1 Contributions

This dissertation makes a number of contributions. I divide them up here according to the area they contribute in.

15.1.1 Better Higher-Order Module Systems

I began this dissertation work with the following thesis:

> By basing a module system on a new type-theoretic concept, namely a new kind of weak sum that can contain both transparent and opaque type definitions called a translucent sum, it is possible to obtain a higher-order module system with the advantages of both the opaque and the transparent approaches, but none of their disadvantages.

I believe I have by now firmly established this thesis. In chapter 3, I showed how my approach using translucent sums provides the advantages of the opaque approach, namely

- (Open) data abstraction is provided at the module level.
- Separate compilation is supported.
- Modules are first-class values.

as well as the advantages of the transparent approach, namely

- Modules can "contain" other types.
- Modules may export type abbreviations.

Moreover, my approach provides the following additional advantages not available in either of the two previous approaches:

- All of the higher-order idioms work.
- Transparency and opacity can be mixed, even in the same module.

My approach also offers the following advantages:

- User-defined higher-order constructors are available.
- Functors are first-class.
- Higher-order functors are provided.
- Type-sharing specifications can be encoded.
- Linking and separate compilation can be described in the resulting programming language.
- The resulting system has a simple and uniform type theory.

My approach thus represents a major step forward in the design of higher-order module systems over previous approaches, providing the first truly satisfactory higher-order module system. This is the single biggest contribution of my dissertation.

15.1.2 Feasibility of My Approach

In order to show that these advantages are actually realizable in practice, I need to show that my approach is feasible. I have done this by creating a complete working system using my approach — the kernel system — and proving that it has the necessary properties. In particular, I have shown the following:

- How to arrange the system and its associated proofs so they are tractable
- The system's soundness even in the presence of side effects
- How to do type checking effectively by giving:
  - a recursive algorithm for constructor and assignment validity
  - a recursive algorithm for constructor equality

- a semi-decision procedure for subtyping
- a proof that type checking is decidable, given an oracle for subtyping
- a semi-decision procedure for type checking
- an argument that the semi-decidability of type checking is not likely to be problem in practice

All of these represent contributions of my dissertation; the first item above is particularly important because of the complexity of systems providing translucent sums. I expect that this dissertation, especially the section on recommended improvements (Section 9.7), will prove invaluable to future designers of systems with translucent sums. In additional to being useful as a starting point for building extended systems, my kernel system is likely to be useful as a reduction target because of these proven properties; for example, I expect that the soundness of Xavier Leroy's system [31] can be established in this way far more easily than if it was proved directly.

The negative results about soundness under certain extensions, the decidability of subtyping (and hence type checking), and the existence of principle types if implicit subsumption is permitted are also contributions of my dissertation. Similarly to the case with the full kernel system, I expect that the simple system (see Section 10.6) will be useful in proving subtyping in other systems with translucent sums undecidable.

15.1.3 Technical Contributions

My dissertation also contains a number of more technical contributions. The most obvious of these are a large number of properties about the kernel system's type system that are useful in reasoning about the kernel system. For example, I have shown that in the kernel system all of the following operations preserve judgment validity:

- weakening (adding extra declarations to assignments)
- strengthening (removing unreferenced declarations from assignments)
- replacing a type in an assignment by a subtype
- subject reduction on constructors

Several of the techniques I used to make the proofs more tractable are also noteworthy:

- Specializing the well-typed term judgment to places so that the constructor validity and equality judgments do not depend on term validity or subtyping judgments
- Using the self function to normalize typing derivations

- Defining subtyping as a series of equality and one-step subtyping steps
- Factoring translucent sums into dependent sums plus reified constructors
- Introducing a separate tagged system to remove the dependency of rewriting (and hence equality) on assignments

The first three of these techniques are tied fairly closely to systems involving translucent sums. However, the fourth technique, factoring, could be used to simplify any system with modules or sums that allow fields to contain either terms or types, and the fifth technique, the use of a tagged system, could be applied usefully in any system where the rewriting of constructors depends on the assignment.

15.2 Future work

One obvious piece of future work would be to construct a prototype based on the kernel system, but with a more programmer-friendly interface. Ideally, some form of type inference would be devised for the prototype so as to reduce the amount of type information that would need to be given by the programmer. The prototype would be especially useful in evaluating the value of having modules as first-class values; exploration with the prototype might well lead to the discovery of useful new higher-order–module idioms.

The two extensions to the kernel system that I think are the most interesting to pursue are adding reordering via an elaboration stage (this will require determining the proper reordering rule(s) for subtyping), and adding transparent value declarations to permit inlining. I also think that it is worth investigating the relationship between first-class modules and objects in more detail.

Bibliography

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[2] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[3] Richard J. Beach. Experience with the Cedar programming environment for computer graphics research. Technical Report CSL-84-6, Xerox Corporation, Palo Alto, July 1985.

[4] Edoardo Biagioni. A structured TCP in Standard ML. In *Sigcomm '94*, London, England, August/September 1994.

[5] Edoardo Biagioni, Robert Harper, and Peter Lee. Standard ML signatures for a protocol stack. Technical Report CMU–CS–93–170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1993. (Also published as Fox Memorandum CMU–CS–FOX–93–01).

[6] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *ACM Conference on LISP and Functional Programming*, Orlando, Florida, June 1994.

[7] Hans-Jürgen Böhm, Alan Demers, and James Donahue. An informal description of Russell. Technical Report 80–430, Computer Science Department, Cornell University, Ithaca, New York, 1980.

[8] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In Kahn et al. [28], pages 1–50.

[9] Luca Cardelli. Structural subtyping and the notion of power type. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, 1988. ACM.

[10] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[11] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report 56, DEC Systems Research Center, Palo Alto, CA, March 1990.

[12] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[13] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. Early experience with Mesa. Technical Report CSL-76-6, Xerox Corporation, Palo Alto, October 1976.

[14] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduuction in typed λ-calculi. In *Seventh Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992.

[15] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

[16] Robert Harper. Personal communication, 1993.

[17] Robert Harper. A propsoal for ML2000 (draft of december 16, 1994). (Unpublished), December 1994.

[18] Robert Harper, Peter Lee, Frank Pfenning, and Eugene Rollins. Incremental recompilation for Standard ML of New Jersey. Technical Report CMU–CS–94–116, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1994. (Also published as Fox Memorandum CMU–CS–FOX–94-02; to appear *Workshop on ML*, Orlando, FL, June, 1994.).

[19] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, SC, January 1993. ACM, ACM.

[20] 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.

[21] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML (version 3). Technical Report ECS–LFCS–89–81, Laboratory for the Foundations of Computer Science, Edinburgh University, May 1989.

[22] 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.

[23] 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.

[24] Robert Harper and Chris Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU–CS–96–136R, Carnegie Mellon University, Pittsburgh, PA, 1996.

[25] James G. Hook. Understanding Russell: A first attempt. In Kahn et al. [28], pages 69–85.

[26] K. Jensen. *Pascal User Manual and Report*. Springer-Verlag, 1978. (2nd edition).

[27] Mark P. Jones. Using parameterized signatures to express modular structure. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 21–24, St. Petersburg Beach, FL, January 1996. ACM Press.

[28] Gilles Kahn, David MacQueen, and Gordon Plotkin, editors. *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1984.

[29] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-hall, Inc., 1978.

[30] Butler W. Lampson. A description of the Cedar language: A Cedar language reference manual. Technical Report CSL-83-15, Xerox Corporation, Palo Alto, December 1983.

[31] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland*, pages 109–122. ACM, January 1994.

[32] 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.

[33] Xavier Leroy. A modular module system. Research report 2866, INRIA, April 1996.

[34] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[35] Mark Lillibridge, Zhong Shao, and Robert Harper. Minutes of the ML2000 meeting at Portland, Oregon on January 16, 1994. Unpublished, 1994.

[36] Barbara Liskov, Russell Atkinson, et al. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[37] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[38] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In D. Sannella, editor, *Programming Languages and Systems — ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

[39] David C. J. Matthews. POLY report. Technical Report 28, Computer Laboratory, University of Cambridge, 1982.

[40] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL-78-1, Xerox Corporation, Palo Alto, February 1978.

[41] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[42] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

[43] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. In *Twelfth ACM Symposium on Principles of Programming Languages*, 1985.

[44] John Gregory Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. (Available as Carnegie Mellon University School of Computer Science technical report CMU–CS–95–226.).

[45] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[46] John Ophel. A polymorphic language with first-class modules. *Australian Computer Science Communications*, 17(1):422–430, February 1995.

[47] Amit Patel and Chris Stone. Minutes of the ML2000 meeting at King's Beach, CA. (Unpublished), August 1996.

[48] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[49] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359. Springer-Verlag LNCS 352, March 1989.

[50] Benjamin Pierce. Bounded quantification is undecidable. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, Albuquerque*. ACM, January 1992.

[51] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1991.

[52] I. C. Pyle. *The Ada Programming Language*. Prentice-Hall International, 1981.

[53] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1987.

[54] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. Technical Report CSL-86-1, Xerox Corporation, Palo Alto, June 1986.

[55] Warren Teitelman. The Cedar programming environment: A midterm report and examination. Technical Report CSL-83-11, Xerox Corporation, Palo Alto, June 1984.

[56] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1983.