# Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

# Chapter 8: Modules

## Overview

The language features we have introduced so far are very powerful for building systems of a few hundred lines of code. If we are to build larger systems, with thousands of lines of code, we will need some more ingredients.

1. We will need a good way to separate the system into relatively self-contained parts, and to document the dependencies between those parts.

2. We will need a better way to control the scope and binding of names. Lexical scoping is a powerful tool for name control, but it is not sufficient when programs may be large or split up over multiple sources.

3. We will need a way to enforce abstraction boundaries. In chapter 2, we introduced the idea of an abstract data type. Inside the implementation of the type, we can manipulate the values arbitrarily, but outside the implementation, the values of the type are to be created and manipulated only by the procedures in the interface of that type. We call this an *abstraction boundary*. If a program respects this boundary, we can change the implementation of the data type. If, however, some piece of code breaks the abstraction by relying on the details of the implementation, then we can no longer change the implementation freely without breaking other code.

4. Last, we need a way to combine these parts flexibly, so that a single part may be reused in different contexts.

In this chapter, we introduce *modules* as a way of satisfying these needs. In particular, we show how we can use the type system to create and enforce abstraction boundaries.

A program in our module language consists of a sequence of *module definitions* followed by an expression to be evaluated. Each module definition binds a name to a *module*. A created module is either a *simple module*, which is a set of bindings, much like an environment, or a *module procedure* that takes a module and produces another module.

Each module will have an *interface*. A module that is a set of bindings will have a *simple interface*, which lists the bindings offered by the module, and their types. A module procedure will have an interface that specifies the interfaces of the argument and result modules of the procedure, much as a procedure has a type that specifies the types of its argument and result.

These interfaces, like types, determine the ways in which modules can be combined. We therefore emphasize the types of our examples, since evaluation of these programs is straightforward. As we have seen before, understanding the scoping and binding rules of the language will be the key to both analyzing and evaluating programs in the language.

## 8.1 The Simple Module System

Our first language, SIMPLE-MODULES, has only simple modules. It does not have module procedures, and it creates only very simple abstraction boundaries. This module system is similar to that used in several popular languages.

### 8.1.1 Examples

Imagine a software project involving three developers: Alice, Bob, and Charlie. Alice, Bob, and Charlie are developing largely independent pieces of the project. These developers are geographically dispersed, perhaps in different time zones. Each piece of the project is to implement an interface, like those in section 2.1, but the implementation of that interface may involve a large number of additional procedures. Furthermore, each of the developers needs to make sure that there are no name conflicts that would interfere with the other portions of the project when the pieces are integrated.

To accomplish this goal, each of the developers needs to publish an interface, listing the names for each of their procedures that they expect others to use. It will be the job of the module system to ensure that these names are public, but any other names they use are private and will not be overridden by any other piece of code in the project.

We could use the scoping techniques of chapter 3, but these do not scale to larger projects. Instead, we will use a module system. Each of our developers will produce a module consisting of a public interface and a private implementation. Each developer can see the interface and implementation of his or her own module, but Alice can see only the interfaces of the other modules. Nothing she can do can interfere with the implementations of the other modules, nor can their module implementations interfere with hers. (See figure 8.1.)

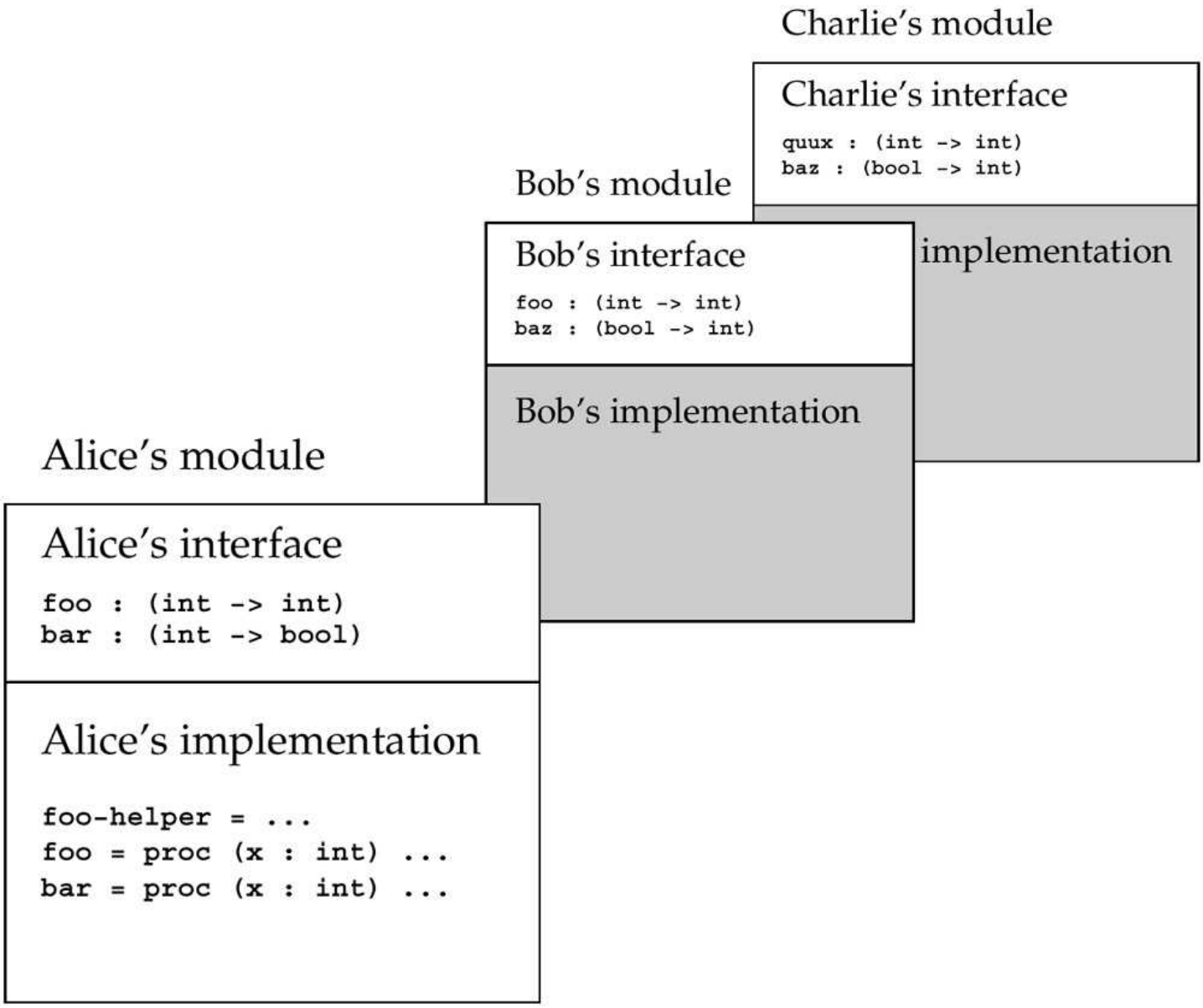


Figure 8.1: Alice's view of the three modules in the project

Here is a short example in SIMPLE-MODULES.

Example 8.1

```
module m1
 interface
  [a : int
   b : int
   c : int]
 body
  [a = 33
   x = -(a,1)  % = 32
   b = -(a,x)  % =  1
   c = -(x,b)] % = 31
let a = 10
in -(-(from m1 take a,
       from m1 take b),
     a)
```

has type int and value ((33 – 1) – 10) = 22.

This program begins with the definition of a module named m1. Like all modules, it has an *interface* and a *body*. The body *implements* the interface. The interface *declares* the variables a, b, and c. The body *defines* bindings for a, x, b, and c.

When we evaluate the program, the expressions in m1's body are evaluated. The appropriate values are bound to the variables from m1 take a, from m1 take b, and from m1 take c, which are in scope after the module definition. from m1 take x is not in scope after the module definition, since it has not been declared in the interface.

These new variables are called *qualified variables* to distinguish them from our previous *simple variables*. In conventional languages, qualified variables might be written m1.a or m1:a or m1::a. The notation m1.a is often used for something different in object-oriented languages, which we study in chapter 9.

We say that the interface *offers* (or *advertises* or *promises*) three integer values, and that the body *supplies* (or *provides* or *exports*) these values. A module body *satisfies* an interface when it supplies a value of the advertised type for each of the variables that are named in the interface.

In the body, definitions have let* scoping, so that a is in scope in the definitions of x, b, and c. Some of the scopes are pictured in figure 8.2.

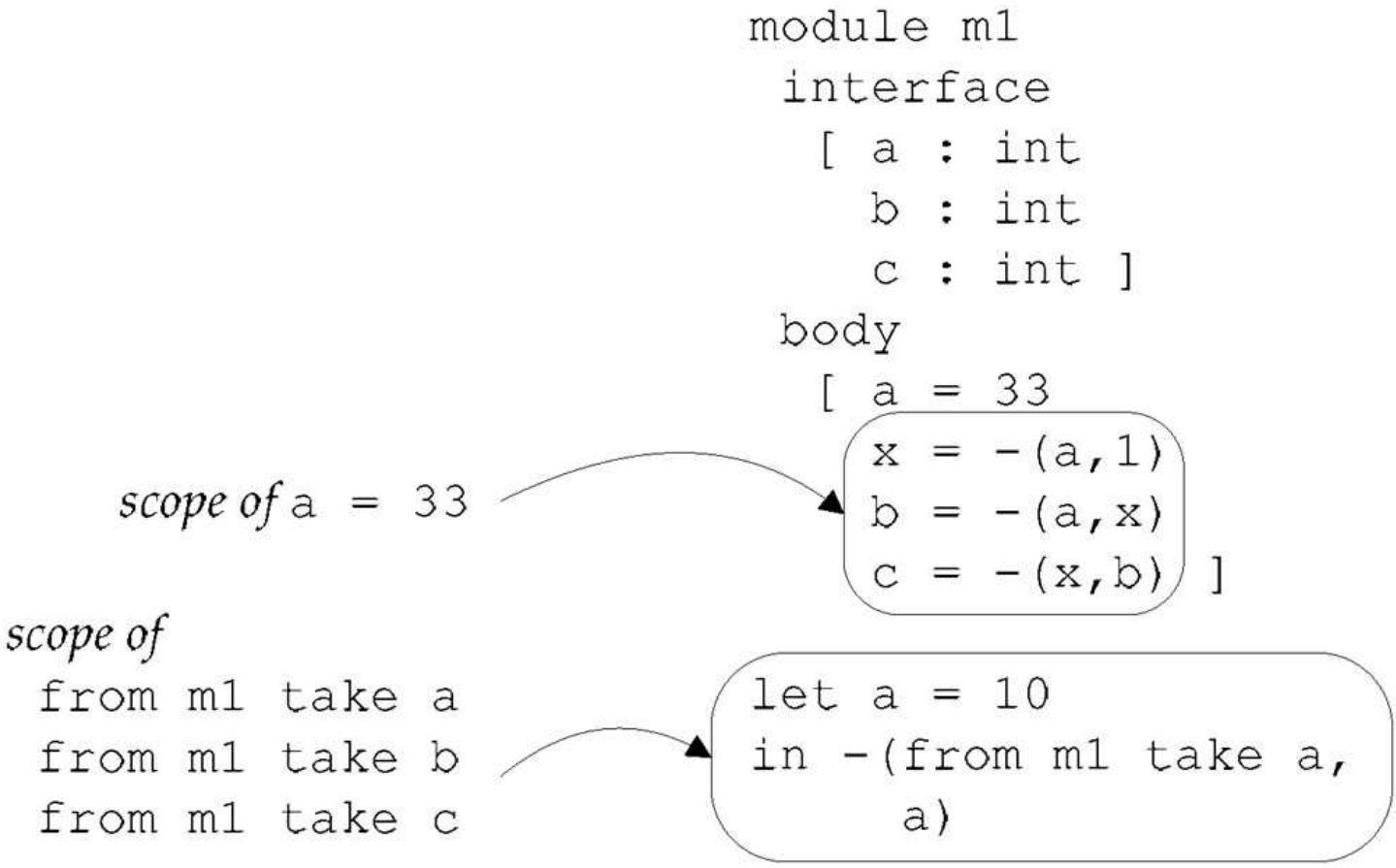


Figure 8.2: Some of the scopes for a simple module

In this example, the expression, starting with let a = 10, is the *program body*. Its value will become the value of the program.

Each module establishes an abstraction boundary between the module body and the rest of the program. The expressions in the module body are *inside* the abstraction boundary, and everything else is *outside* the abstraction boundary. A module body may supply bindings for names that are not in the interface, but those bindings are not visible in the program body or in other modules, as suggested in figure 8.1. In our example, from m1 take x is not in scope. Had we written -(from m1 take a, from m1 take x), the resulting program would have been ill-typed.

Example 8.2: The program

```
module m1
 interface
  [u : bool]
 body
  [u = 33]

44
```

is not well-typed. The body of the module must associate each name in the interface with a value of the appropriate type, even if those values are not used elsewhere in the program.

Example 8.3: The module body must supply bindings for all the declarations in the interface. For example,

```
module m1
 interface
  [u : int
   v : int]
 body
  [u = 33]

44
```

is not well-typed, because the body of m1 does not provide all of the values that its interface advertises.

Example 8.4: To keep the implementation simple, our language requires that the module body produce the values in the same order as the interface. Hence

```
module m1
 interface
  [u : int
   v : int]
 body
  [v = 33
   u = 44]

from m1 take u
```

is not well-typed. This can be fixed (exercises 8.8, 8.17).

Example 8.5: In our language, modules have `let`* scoping (exercise 3.17). For example,

```
module m1
 interface
  [u : int]
 body
  [u = 44]

module m2
 interface
  [v : int]
 body
  [v = -(from m1 take u,11)]

-(from m1 take u, from m2 take v)
```

has type int. But if we reverse the order of the definitions, we get

```
module m2
 interface
  [v : int]
 body
  [v = -(from m1 take u,11)]

module m1
 interface
  [u : int]
 body
  [u = 44]

-(from m1 take u, from m2 take v)
```

which is not well-typed, since from m1 take u is not in scope where it is used in the body of m2.

### 8.1.2 Implementing the Simple Module System

## Syntax

A program in SIMPLE-MODULES consists of a sequence of module definitions, followed by an expression.

$$program ::= \{ModuleDefn\}^* \ Expression$$

```
a-program (m-defs body)
```

A module definition consists of its name, its interface, and its body.

$$ModuleDefn ::= \texttt{module}\ Identifier\ \texttt{interface}\ Iface\ \texttt{body}\ ModuleBody$$

```
a-module-definition (m-name expected-iface m-body)
```

An interface for a simple module consists of an arbitrary number of declarations. Each declaration declares a program variable and its type. We call these *value declarations*, since the variable being declared will denote a value. In later sections, we introduce other kinds of interfaces and declarations.

$$Iface ::= [\ \{Decl\}^*\ ]$$

```
simple-iface (decls)
```

$$Decl ::= Identifier\ :\ Type$$

```
val-decl (var-name ty)
```

A module body consists of an arbitrary number of definitions. Each definition associates a variable with the value of an expression.

$$ModuleBody ::= [\ \{Defn\}^*\ ]$$

```
defns-module-body (defns)
```

$$Defn ::= Identifier\ =\ Expression$$

```
val-defn (var-name exp)
```

Our expressions are those of CHECKED (section 7.3), but we modify the grammar to add a new kind of expression for a reference to a qualified variable.

$$Expression ::= \texttt{from}\ Identifier\ \texttt{take}\ Identifier$$

```
qualified-var-exp (m-name var-name)
```

### The Interpreter

Evaluation of a module body will produce a *module*. In our simple module language, a module will be an environment consisting of all the bindings exported by the module. We represent these with the data type typed-module.

```
(define-datatype typed-module typed-module?
  (simple-module
    (bindings environment?)))
```

We bind module names in the environment, using a new kind of binding:

```
(define-datatype environment environment?
  (empty-env)
  (extend-env ...as before...)
  (extend-env-rec ...as before...)
  (extend-env-with-module
    (m-name symbol?)
    (m-val typed-module?)
    (saved-env environment?)))
```

For example, if our program is

```
module m1
 interface
  [a : int
   b : int
   c : int]
 body
  [a = 33
   b = 44
   c = 55]
module m2
 interface
  [a : int
   b : int]
 body
  [a = 66
   b = 77]
let z = 99
in -(z, -(from m1 take a, from m2 take a))
```

then the environment after the declaration of z is

```
#(struct:extend-env
   z #(struct:num-val 99)
  #(struct:extend-env-with-module
     m2 #(struct:simple-module
           #(struct:extend-env
              a #(struct:num-val 66)
              #(struct:extend-env
                 b #(struct:num-val 77)
                 #(struct:empty-env))))
     #(struct:extend-env-with-module
        m1 #(struct:simple-module
              #(struct:extend-env
                 a #(struct:num-val 33)
                 #(struct:extend-env
                    b #(struct:num-val 44)
                    #(struct:extend-env
                       c #(struct:num-val 55)
                       #(struct:empty-env)))))
       #(struct:empty-env))))
```

In this environment, both m1 and m2 are bound to simple modules, which contain a small environment.

To evaluate a reference to a qualified variable from *m* take *var*, we use lookup-qualified-var-in-env. This first looks up the module *m* in the current environment, and then looks up *var* in the resulting environment.

```
lookup-qualified-var-in-env : Sym × Sym × Env → ExpVal
(define lookup-qualified-var-in-env
  (lambda (m-name var-name env)
    (let ((m-val (lookup-module-name-in-env m-name env)))
      (cases typed-module m-val
        (simple-module (bindings)
          (apply-env bindings var-name))))))
```

To evaluate a program, we evaluate its body in an initial environment built by adding all the module definitions to the environment. The procedure add-module-defns-to-env loops through the module definitions. For each module definition, the body is evaluated, and the resulting module is added to the environment. See figure 8.3.

---

```
value-of-program : Program → ExpVal
(define value-of-program
  (lambda (pgm)
```

```
    (cases program pgm
      (a-program (m-defns body)
        (value-of body
          (add-module-defns-to-env m-defns (empty-env)))))))

add-module-defns-to-env : Listof(Defn) × Env → Env
(define add-module-defns-to-env
  (lambda (defns env)
    (if (null? defns)
      env
      (cases module-definition (car defns)
        (a-module-definition (m-name iface m-body)
          (add-module-defns-to-env
            (cdr defns)
            (extend-env-with-module
              m-name
              (value-of-module-body m-body env)
              env)))))))
```

Figure 8.3: Interpreter for SIMPLE-MODULES, part 1

Last, to evaluate a module body, we build an environment, evaluating each expression in the appropriate environment to get let* scoping. The procedure defns-to-env produces an environment containing only the bindings produced by the definitions defns (figure 8.4).

```
value-of-module-body : ModuleBody × Env → TypedModule
(define value-of-module-body
  (lambda (m-body env)
    (cases module-body m-body
      (defns-module-body (defns)
        (simple-module
          (defns-to-env defns env))))))

defns-to-env : Listof(Defn) × Env → Env
(define defns-to-env
  (lambda (defns env)
    (if (null? defns)
      (empty-env)
      (cases definition (car defns)
        (val-defn (var exp)
          (let ((val (value-of exp env)))
            (let ((new-env (extend-env var val env)))
              (extend-env var val
                (defns-to-env
                  (cdr defns) new-env)))))))))
```

Figure 8.4: Interpreter for SIMPLE-MODULES, part 2

## The Checker

The job of the checker is to make sure that each module body satisfies its interface, and that each variable is used consistently with its type.

The scoping rules of our language are fairly simple: Modules follow let* scoping, putting into scope qualified variables for each of the bindings exported by the module. The interface tells us the type of each qualified variable. Declarations and definitions both follow let* scoping as well (see figure 8.2).

As we did with the checker in chapter 7, we use the type environment to keep track of information about each name that is in scope. Since we now have module names, we bind module names in the type environment. Each module name will be bound to its interface, which plays the role of a type.

```
(define-datatype type-environment type-environment?
  (empty-tenv)
  (extend-tenv ...as before...)
  (extend-tenv-with-module
    (name symbol?)
    (interface interface?)
    (saved-tenv type-environment?)))
```

We find the type of a qualified variable from *m* take *var* by first looking up *m* in the type environment, and then looking up the type of *var* in the resulting interface.

```
lookup-qualified-var-in-tenv : Sym × Sym × Tenv → Type
(define lookup-qualified-var-in-tenv
  (lambda (m-name var-name tenv)
    (let ((iface (lookup-module-name-in-tenv tenv m-name)))
      (cases interface iface
        (simple-iface (decls)
          (lookup-variable-name-in-decls var-name decls))))))
```

---

```
type-of-program : Program → Type
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (module-defns body)
        (type-of body
          (add-module-defns-to-tenv module-defns
            (empty-tenv)))))))

add-module-defns-to-tenv : Listof(ModuleDefn) × Tenv → Tenv
(define add-module-defns-to-tenv
  (lambda (defns tenv)
    (if (null? defns)
      tenv
      (cases module-definition (car defns)
        (a-module-definition (m-name expected-iface m-body)
          (let ((actual-iface (interface-of m-body tenv)))
            (if (<:-iface actual-iface expected-iface tenv)
            (let ((new-tenv
                    (extend-tenv-with-module
                      m-name
                      expected-iface
                      tenv)))
                (add-module-defns-to-tenv
                  (cdr defns) new-tenv))
              (report-module-doesnt-satisfy-iface
                m-name expected-iface actual-iface)))))))))
```

---

Figure 8.5: Checker for SIMPLE-MODULES, part 1

Just as in chapter 7, the process of typechecking a program mimics the evaluation of the program, except that instead of keeping track of values, we keep track of types. Instead of value-of-program, we have type-of-program, and instead of add-module-defns-to-env, we have add-module-defns-to-tenv. The procedure add-module-defns-to-tenv checks each module to see whether the interface produced by the module body matches the advertised interface, using the procedure <:-iface. If it does, the module is added to the type environment. Otherwise, an error is reported.

The interface of a module body associates each variable defined in the body with the type of its definition. For example, if we looked at the body from our first example,

```
[a = 33
 x = -(a,1)
 b = -(a,x)
 c = -(x,b)]
```

we should get

```
[a : int
 x : int
 b : int
 c : int]
```

Once we build an interface describing all the bindings exported by the module body, we can compare it to the interface that the module advertises.

Recall that a simple interface contains a list of declarations. The procedure defns-to-decls creates such a list, calling type-of to find the type of each definition. At every step it also extends the local type environment, to follow the correct let* scoping. (See figure 8.6.)

---

```
interface-of : ModuleBody × Tenv → Iface
(define interface-of
  (lambda (m-body tenv)
    (cases module-body m-body
      (defns-module-body (defns)
        (simple-iface
          (defns-to-decls defns tenv))))))

defns-to-decls : Listof(Defn) × Tenv → Listof(Decl)
(define defns-to-decls
  (lambda (defns tenv)
    (if (null? defns)
      '()
      (cases definition (car defns)
        (val-defn (var-name exp)
          (let ((ty (type-of exp tenv)))
            (cons
              (val-decl var-name ty)
              (defns-to-decls
                (cdr defns)
                (extend-tenv var-name ty tenv)))))))))
```

Figure 8.6: Checker for SIMPLE-MODULES, part 2

All that's left is to compare the actual and expected types of each module, using the procedure <:-iface. We intend to define <: so that if $i_1$ <: $i_2$, then any module that satisfies interface $i_1$ also satisfies interface $i_2$. For example

```
[u : int       [u : int
 v : bool  <:   z : int]
 z : int]
```

since any module that satisfies the interface [u : int v : bool z : int] provides all the values that are advertised by the interface [u : int z : int].

For our simple module language, <:-iface just calls <:-decls, which compares declarations. These procedures take a tenv argument that is not used for the simple module system, but will be needed in section 8.2. See figure 8.7.

```
<:-iface : Iface × Iface × Tenv → Bool
(define <:-iface
  (lambda (iface1 iface2 tenv)
    (cases interface iface1
      (simple-iface (decls1)
        (cases interface iface2
          (simple-iface (decls2)
            (<:-decls decls1 decls2 tenv)))))))

<:-decls : Listof(Decl) × Listof(Decl) × Tenv → Bool
(define <:-decls
  (lambda (decls1 decls2 tenv)
    (cond
      ((null? decls2) #t)
      ((null? decls1) #f)
      (else
        (let ((name1 (decl->name (car decls1)))
              (name2 (decl->name (car decls2))))
          (if (eqv? name1 name2)
            (and
              (equal?
                (decl->type (car decls1))
                (decl->type (car decls2)))
              (<:-decls (cdr decls1) (cdr decls2) tenv))
            (<:-decls (cdr decls1) decls2 tenv)))))))
```

Figure 8.7: Comparing interfaces for SIMPLE-MODULES

The procedure <:-decls does the main work of comparing two sets of declarations. If $decls_1$ and $decls_2$ are two sets of declarations, we say $decls_1$ <: $decls_2$ if and only if any module that supplies bindings for the declarations in $decls_1$ also supplies bindings for the declarations in $decls_2$. This can be assured if $decls_1$ contains a matching declaration for every declaration in $decls_2$, as in the example above.

The procedure <:-decls first checks decls1 and decls2. If decls2 is empty, then it makes no demands on decls1, so the answer is #t. If decls2 is non-empty, but decls1 is empty, then decls2 requires something, but decls1 has nothing. So the answer is #f. Otherwise, we compare the names of the first variables declared by decls1 and decls2. If they are the same, then their types must match, and we recur on the rest of both lists of declarations. If they are not the same, then we recur on the cdr of decls1 to look for something that matches the first declaration of decls2.

This completes the simple module system.

Exercise 8.1 [*] Modify the checker to detect and reject any program that defines two modules with the same name.

Exercise 8.2 [*] The procedure add-module-defn-to-env is not quite right, because it adds all the values defined by the module, not just the ones in the interface. Modify add-module-defn-to-env so that it adds to the environment only the values declared in the interface. Does add-module-defn-to-tenv suffer from the same problem?

Exercise 8.3 [*] Change the syntax of the language so that a qualified variable reference appears as m.v, rather than from m take v.

Exercise 8.4 [*] Change the expression language to include multiple let declarations, multiargument procedures, and multiple letrec declarations, as in exercise 7.24.

Exercise 8.5 [*] Allow let and letrec declarations to be used in module bodies. For example, one should be able to write

```
module even-odd
 interface
  [even : int -> bool
   odd : int -> bool]
 body
  letrec
   bool local-odd (x : int)  = ... (local-even -(x,1)) ...
   bool local-even (x : int) = ... (local-odd -(x,1)) ...
  in [even = local-even
      odd = local-odd]
```

Exercise 8.6 [**] Allow local module definitions to appear in module bodies. For example, one should be able to write

```
module m1
 interface
  [u : int
   v : int]
 body
  module m2
   interface [v : int]
   body [v = 33]
  [u = 44
   v = -(from m2 take v, 1)]
```

Exercise 8.7 [**] Extend your solution to the preceding exercise to allow modules to export other modules as components. For example, one should be able to write

```
module m1
 interface
  [u : int
   n : [v : int]]
 body
  module m2
   interface [v : int]
   body [v = 33]
  [u = 44
   n = m2]

from m1 take n take v
```

Exercise 8.8 [**] In our language, the module must produce the values in the same order as the interface, but that could easily be fixed. Fix it.

Exercise 8.9 [**] We said that our module system should document the dependencies between modules. Add this capability to SIMPLE-MODULES by requiring a depends-on clause in each module body and in the program body. Rather than having all preceding modules in scope in a module m, a preceding module is in scope only if it is listed in m's depends-on clause. For example, consider the program

```
module m1 ...
module m2 ...
module m3 ...
module m4 ...
module m5
 interface [...]
 body
  depends-on m1, m3
  [...]
```

In the body of m5, qualified variables would be in scope only if they came from m1 or m3. A reference to from m4 take x would be ill-typed, even if m4 exported a value for x.

Exercise 8.10 [***] We could also use a feature like depends-on to control when module bodies are evaluated. Add this capability to SIMPLE-MODULES by requiring an imports clause to each module body and program body. imports is like depends-on, but has the additional property that the body of a module is evaluated only when it is imported by some other module (using an imports clause).

Thus if our language had print expressions, the program

```
module m1
 interface [] body [x = print(1)]
module m2
 interface [] body [x = print(2)]
module m3
 interface []
 body
  import m2
  [x = print(3)]
import m3, m1
33
```

would print 2, 3, and 1 before returning 33. Here the modules have empty interfaces, because we are only concerned with the order in which the bodies are evaluated.

Exercise 8.11 [***] Modify the checker to use INFERRED as the language of expressions. For this exercise you will need to modify <:-decls to use something other than equal? to compare types. For example, in

```
module m
 interface [f : (int -> int)]
 body [f = proc (x : ?) x]
```

the actual type for f reported by the type inference engine will be something like (tvar07 -> tvar07), and this should be accepted. On the other hand, we should reject the module

```
module m
 interface [f : (int -> bool)]
 body [f = proc (x : ?) x]
```

even though the type inference engine will report the same type (tvar07 -> tvar07) for f.

## 8.2 Modules That Declare Types

So far, our interfaces have declared only ordinary variables and their types. In the next module language, OPAQUE-TYPES, we allow interfaces to declare types as well. For example, in the definition

```
module m1
 interface
  [opaque t
   zero : t
   succ : (t -> t)
   pred : (t -> t)
   is-zero : (t -> bool)]
 body
  ...
```

the interface declares a type t, and some operations zero, succ, pred, and is-zero that operate on values of that type. This is the interface that might be associated with an implementation of arithmetic, as in section 2.1. Here t is declared to be an *opaque type*, meaning that code outside the module boundary does not know how values of this type are represented. All the outside code knows is that it can manipulate values of type from m1 take t with the procedures from m1 take zero, from m1 take succ, etc. Thus from m1 take t behaves like a primitive type such as int or bool.

We will introduce two kinds of type declarations: *transparent* and *opaque*. Both are necessary for a good module system.

### 8.2.1 Examples

To motivate this, consider our developers again. Alice has been using a data structure consisting of a pair of integers, representing the x- and y-coordinates of a point. She is using a language with types like those of exercise 7.8, so her module, named Alices-points, has an interface with declarations like

```
initial-point : (int -> pairof int * int)
increment-x   : (pairof int * int -> pairof int * int)
```

Bob and Charlie complain about this. They don't want to have to write pairof int * int over and over again. Alice therefore rewrites her interface to use transparent type declarations. This allows her to write

```
module Alices-points
 interface
  [transparent point = pairof int * int
   initial-point : (int -> point)
   increment-x   : (point -> point)
   get-x         : (point -> int)
   ...]
```

This simplifies her task, since she has less writing to do, and it makes her collaborators' tasks simpler, because in their implementations they can write definitions like

```
[transparent point = from Alices-points take point
 foo = proc (p1 : point)
        proc (p2 : point) ...
 ...]
```

For some projects, this would do nicely. On the other hand, the points in Alice's project happen to represent points on a metal track with a fixed geometry, so the x- and y-coordinates are not independent. Alice's implementation of increment-x carefully updates the y-coordinate to match the change in the x-coordinate. But Bob doesn't know this, and so he writes his own procedure

```
increment-y = proc (p : point)
               unpair x y = p
               in newpair(x, -(y,-1))
```

Because Bob's code changes the y-coordinate without changing the x-coordinate correspondingly, Alice's code no longer works correctly.

Worse yet, what if Alice decides to change the representation of points so that the y-coordinate is in the first component? She can change her code to match this new representation. But then Bob's code would be broken, because his increment-y procedure now changes the wrong component of the pair.

Alice can solve her problem by making point an *opaque* data type. She rewrites her interface to say

```
opaque point
initial-point : (int -> point)
increment-x   : (point -> point)
get-x         : (point -> int)
```

Now Bob can create new points using the procedure initial-point, and he can manipulate points using from Alices-points take get-x and from Alices-points take increment-x, but he can no longer manipulate points using any procedures other than the ones in Alice's interface. In particular, he can no longer write the increment-y procedure, since it manipulates a point using something other than the procedures in Alice's interface.

In the remainder of this section, we explore further examples of these facilities.

### Transparent Types

We begin by discussing *transparent* type declarations. These are sometimes called *concrete* type declarations or *type abbreviations*.

has type (int -> bool).

The declaration transparent t = int in the interface binds t to the type int in the rest of the interface, so we can write z:t. More importantly, it also binds from m1 take t to int in the rest of the program. We call this a *qualified type*. Here we have used it to declare the type of the bound variable z. The scope of a declaration is the rest of the interface and the rest of the program after the module definition.

The definition type t = int in the body binds t to the type int in the rest of the module body, so we could write s = proc (x : t).... As before, the scope of a definition is the rest of the body (see figure 8.8).

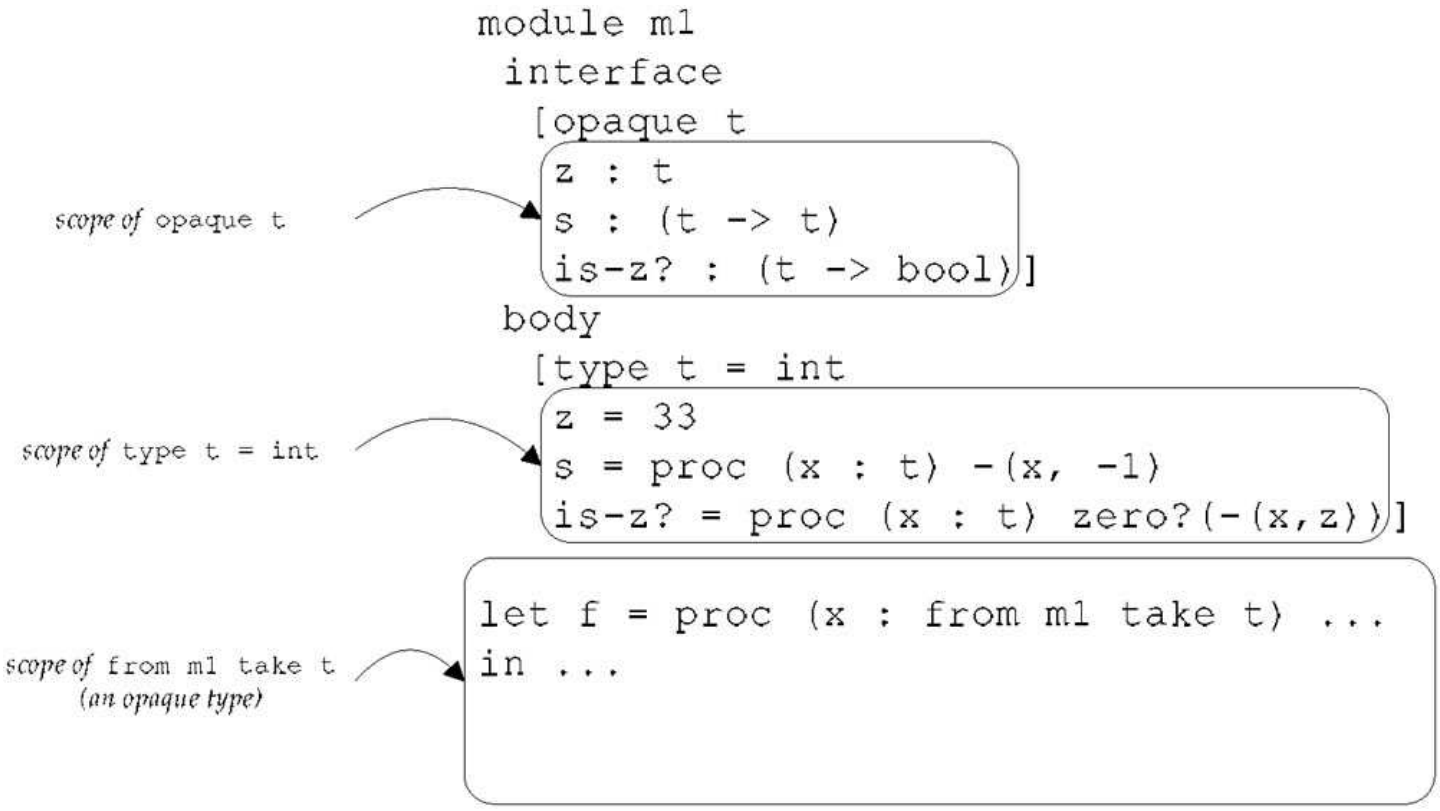


Figure 8.8: Scopes for a module that declares types

Of course, we can use any name we like for the type, and we can declare more than one type. The type declarations can appear anywhere in the interface, so long as each declaration precedes all of its uses.

## Opaque Types

A module can also export *opaque* types by using an opaque-type declaration. Opaque types are sometimes called *abstract* types.

Example 8.7: Let's take the program in example 8.6 and replace the transparent type declaration by an opaque one. The resulting program is

```
module m1
 interface
  [opaque t
   z : t
   s : (t -> t)
   is-z? : (t -> bool)]
 body
  [type t = int
   z = 33
   s = proc (x : t) -(x,-1)
   is-z? = proc (x : t) zero?(-(x,z))]

 proc (x : from m1 take t)
  (from m1 take is-z? -(x,0))
```

Example 8.6: The program

```
module m1
 interface
  [transparent t = int
   z : t
   s : (t -> t)
   is-z? : (t -> bool)]
 body
  [type t = int
   z = 33
   s = proc (x : t) -(x,-1)
   is-z? = proc (x : t) zero?(-(x,z))]

 proc (x : from m1 take t)
  (from m1 take is-z? -(x,0))
```

The declaration opaque t in the interface declares t to be the name of a new opaque type. An opaque type behaves like a new primitive type, such as int or bool. The named type t is bound to this opaque type in the rest of the interface, and the qualified type from m1 take t is bound to the same opaque type in the rest of the program. All the rest of the program knows about the type from m1 take t is that from m1 take z is bound to a value of that type, and that from m1 take s and from m1 take is-z? are bound to procedures that can manipulate values of that type. This is the abstraction boundary. The type checker guarantees that the evaluation of an expression that has type from m1 take t is safe, so that the value of the expression has been constructed only by these operators, as discussed on page 239.

The corresponding definition type t = int defines t to be a name for int inside the module body, but this information is hidden from the rest of the program, because the rest of the program gets its bindings from the module interface.

So -(x, 0) is not well-typed, because the main program does not know that values of type from m1 take t are actually values of type int.

Let's change the program to remove the arithmetic operation, getting

```
module m1
 interface
  [opaque t
   z : t
   s : (t->t)
   is-z? : (t -> bool)]
 body
  [type t = int
   z = 33
   s = proc (x : t) -(x,-1)
   is-z? = proc (x : t) zero?(-(x,z))]

 proc (x : from m1 take t)
  (from m1 take is-z? x)
```

Now we have a well-typed program that has type (from m1 take t -> bool).

By enforcing this abstraction boundary, the type checker guarantees that no program manipulates the values provided by the interface except through the procedures that the interface provides. This gives us a mechanism to enforce the distinction between the users of a data type and its implementation, as discussed in chapter 2. We next show some examples of this technique.

Example 8.8: If a program uses a module definition

```
module colors
 interface
  [opaque color
   red : color
   green : color
   is-red? : (color -> bool)]
 body
  [type color = int
   red = 0
   green = 1
   is-red? = proc (c : color) zero?(c)]
```

there is no way the program can figure out that from colors take color is actually int, or that from colors take green is actually 1

(except, perhaps, by returning a color as the final answer and then printing it out).

Example 8.9: The program

```
module ints1
 interface
  [opaque t
   zero : t
   succ : (t -> t)
   pred : (t -> t)
   is-zero : (t -> bool)]
 body
  [type t = int
   zero = 0
   succ = proc(x : t) -(x,-5)
   pred = proc(x : t) -(x,5)
   is-zero = proc (x : t) zero?(x)]

 let z = from ints1 take zero
 in let s = from ints1 take succ
    in (s (s z))
```

has type from ints1 take t. It has value 10, but we can manipulate this value only through the procedures that are exported from ints1. This module represents the integer *k* by the expressed value 5 * *k*. In the notation of section 2.1, ⌈*k*⌉ = 5 * *k*.

Example 8.10: In this module, ⌈k⌉ = –3 * k.

```
module ints2
 interface
  [opaque t
   zero : t
   succ : (t -> t)
   pred : (t -> t)
   is-zero : (t -> bool)]
 body
  [type t = int
   zero = 0
   succ = proc(x : t) -(x,3)
   pred = proc(x : t) -(x,-3)
   is-zero = proc (x : t) zero?(x)]

 let z = from ints2 take zero
 in let s = from ints2 take succ
    in (s (s z))
```

has type from ints2 take t and has value -6.

Example 8.11: In the preceding examples, we couldn't manipulate the values directly, but we could manipulate them using the procedures exported by the module. As we did in chapter 2, we can compose these procedures to do useful work. Here we combine them to write a procedure `to-int` that converts a value from the module back to a value of type `int`.

```
module ints1 ...as before...

let z = from ints1 take zero
in let s = from ints1 take succ
in let p = from ints1 take pred
in let z? = from ints1 take is-zero
in letrec int to-int (x : from ints1 take t) =
              if (z? x)
              then 0
              else -((to-int (p x)), -1)
in (to-int (s (s z)))
```

has type int and has value 2.

Example 8.12: Here is the same technique used with the implementation of arithmetic `ints2`.

```
module ints2 ...as before...
```

```
let z = from ints2 take zero
in let s = from ints2 take succ
in let p = from ints2 take pred
in let z? = from ints2 take is-zero
in letrec int to-int (x : from ints2 take t)
               = if (z? x)
                 then 0
                 else -((to-int (p x)), -1)
in (to-int (s (s z)))
```

---

also has type int and value 2.

We show in section 8.3 how to abstract over these two examples.

Example 8.13: In the next program, we construct a module to encapsulate a data type of booleans. The booleans are represented as integers, but that fact is hidden from the rest of the program, as in example 8.8.

---

```
module mybool
 interface
  [opaque t
   true : t
   false : t
   and : (t -> (t -> t))
   not:(t->t)
   to-bool : (t -> bool)]
 body
  [type t = int
   true = 0
   false = 13
   and = proc (x : t)
          proc (y : t)
           if zero?(x) then y else false
   not = proc (x : t)
          if zero?(x) then false else true
   to-bool = proc (x : t) zero?(x)]

let true = from mybool take true
in let false = from mybool take false
in let and = from mybool take and
in ((and true) false)
```

---

has type from mybool take t, and has value 13.

Exercise 8.12 [*] In example 8.13, could the definition of and and not be moved from inside the module to outside it? What about to-bool?

Exercise 8.13 [*] Write a module that implements arithmetic using a representation in which the integer *k* is represented as 5 * *k* + 3.

Exercise 8.14 [*] Consider the following alternate definition of mybool (example 8.13):

```
module mybool
 interface
  [opaque t
   true : t
   false : t
   and : (t -> (t -> t))
   not : (t -> t)
   to-bool : (t -> bool)]
 body
  [type t = int
   true = 1
   false = 0
   and = proc (x : t)
          proc (y : t)
           if zero?(x) then false else y
   not = proc (x : t)
          if zero?(x) then true else false
   to-bool = proc (x : t)
              if zero?(x) then zero?(1) else zero?(0)]
```

Is there any program of type int that returns one value using the original definition of mybool, but a different value using the

new definition?

Exercise 8.15 [**] Write a module that implements a simple abstraction of tables. Your tables should be like environments, except that instead of binding symbols to Scheme values, they bind integers to integers. The interface provides a value that represents an empty table and two procedures add-to-table and lookup-in-table that are analogous to extend-env and apply-env. Since our language has only one-argument procedures, we get the equivalent of multiargument procedures by using Currying (exercise 3.20). You may model the empty table with a table that returns 0 for any query. Here is an example using this module.

```
module tables
 interface
  [opaque table
   empty : table
   add-to-table : (int -> (int -> (table -> table)))
   lookup-in-table : (int -> (table -> int))]
 body
  [type table = (int -> int)
   ...]

let empty = from tables take empty
in let add-binding = from tables take add-to-table
in let lookup = from tables take lookup-in-table
in let table1 = (((add-binding 3) 300)
                 (((add-binding 4) 400)
                  (((add-binding 3) 600)
                    empty)))
in -(((lookup 4) table1),
     ((lookup 3) table1))
```

This program should have type int. The table table1 binds 4 to 400 and 3 to 300, so the value of the program should be 100.

### 8.2.2 Implementation

We now extend our system to model transparent and opaque type declarations and qualified type references.

### Syntax and the Interpreter

We add syntax for two new kinds of types: named types (like t) and qualified types (like from m1 take t).

$$\textit{Type} ::= \textit{Identifier}$$

```
named-type (name)
```

$$\textit{Type} ::= \texttt{from}\ \textit{Identifier}\ \texttt{take}\ \textit{Identifier}$$

```
qualified-type (m-name t-name)
```

We add two new kinds of declarations, for opaque and transparent types.

$$\textit{Decl} ::= \texttt{opaque}\ \textit{Identifier}$$

```
opaque-type-decl (t-name)
```

$$\textit{Decl} ::= \texttt{transparent}\ \textit{Identifier}\ \texttt{=}\ \textit{Type}$$

```
transparent-type-decl (t-name ty)
```

We also add a new kind of definition: a type definition. This will be used to define both opaque and transparent types.

$$Defn ::= \texttt{type}\ \ Identifier\ \ \texttt{=}\ \ Type$$

```
type-defn (name ty)
```

The interpreter doesn't look at types or declarations, so the only change to the interpreter is to make it ignore type definitions.

```
defns-to-env : Listof(Defn) × Env → Env
(define defns-to-env
  (lambda (defns env)
    (if (null? defns)
      (empty-env)
      (cases definition (car defns)
        (val-defn (var exp) ...as before...)
        (type-defn (type-name type)
          (defns-to-env (cdr defns) env))))))
```

## The Checker

The changes to the checker are more substantial, since all the manipulations involving types must be extended to handle the new types.

First, we introduce a systematic way of handling opaque and transparent types. An opaque type behaves like a primitive type, such as int or bool. Transparent types, on the other hand, are transparent, as the name suggests: they behave exactly like their definitions. So every type is equivalent to one that is given by the grammar

```
Type ::= int | bool | from m take t | (Type -> Type)
```

where *t* is declared as an opaque type in *m*. We call a type of this form an *expanded type*.

We next extend type environments to handle new types. Our type environments will bind each named type or qualified type to an expanded type. Our new definition of type environments is

```
(define-datatype type-environment type-environment?
  (empty-tenv)
  (extend-tenv ...as before...)
  (extend-tenv-with-module ...as before...)
  (extend-tenv-with-type
    (name type?)
    (type type?)
    (saved-tenv type-environment?)))
```

subject to the condition that type is always an expanded type. This condition is an *invariant*, as discussed on page 10.

We next write a procedure, expand-type, which takes a type and a type environment, and which expands the type using the type bindings in the type environment. It looks up named types and qualified types in the type environment, relying on the invariant that the resulting types are expanded, and for a proc type it recurs on the argument and result types.

```
expand-type : Type × Tenv → ExpandedType
(define expand-type
  (lambda (ty tenv)
    (cases type ty
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (arg-type result-type)
        (proc-type
          (expand-type arg-type tenv)
          (expand-type result-type tenv)))
      (named-type (name)
        (lookup-type-name-in-tenv tenv name))
      (qualified-type (m-name t-name)
        (lookup-qualified-type-in-tenv m-name t-name tenv)))))
```

In order to maintain this invariant, we must be sure to call expand-type whenever we extend the type environment. There are three such places:

- in type-of in the checker,

- where we process a list of definitions, with defns-to-decls, and
- where we add a module to the type environment, in add-module-defns-to-tenv.

In the checker, we replace each call of the form

```
(extend-tenv sym ty tenv)
```

by

```
(extend-tenv var (expand-type ty tenv) tenv)
```

In defns-to-decls, when we encounter a type definition, we expand its right-hand side and add it to the type environment. The type returned by type-of is guaranteed to be expanded, so we don't need to expand it again. We turn a type definition into a transparent type declaration, since in the body all type bindings are transparent. In add-module-defns-to-tenv, we call extend-tenv-with-module, adding an interface to the type environment. In this case we need to expand the interface to make sure that all the types it contains are expanded. To do this, we modify add-module-defns-to-tenv to call expand-iface. See figure 8.9.

---

```
defns-to-decls : Listof(Defn) × Tenv → Listof(Decl)
(define defns-to-decls
  (lambda (defns tenv)
    (if (null? defns)
      '()
      (cases definition (car defns)
        (val-defn (var-name exp)
          (let ((ty (type-of exp tenv)))
            (let ((new-env (extend-tenv var-name ty tenv)))
              (cons
                (val-decl var-name ty)
                (defns-to-decls (cdr defns) new-env)))))
        (type-defn (name ty)
          (let ((new-env
                  (extend-tenv-with-type
                    name (expand-type ty tenv) tenv)))
            (cons
              (transparent-type-decl name ty)
              (defns-to-decls (cdr defns) new-env))))))))

add-module-defns-to-tenv : Listof(ModuleDefn) × Tenv → Tenv
(define add-module-defns-to-tenv
  (lambda (defns tenv)
    (if (null? defns)
      tenv
      (cases module-definition (car defns)
        (a-module-definition (m-name expected-iface m-body)
          (let ((actual-iface (interface-of m-body tenv)))
            (if (<:-iface actual-iface expected-iface tenv)
              (let ((new-env
                      (extend-tenv-with-module m-name
                        (expand-iface
                          m-name expected-iface tenv)
                        tenv)))
                (add-module-defns-to-tenv
                  (cdr defns) new-env))
              (report-module-doesnt-satisfy-iface
                m-name expected-iface actual-iface)))))))))
```

---

Figure 8.9: Checker for OPAQUE-TYPES, part 1

The procedure expand-iface (figure 8.10) calls expand-decls. We separate these procedures in preparation for section 8.3.

---

```
expand-iface : Sym × Iface × Tenv → Iface
(define expand-iface
  (lambda (m-name iface tenv)
    (cases interface iface
      (simple-iface (decls)
        (simple-iface
          (expand-decls m-name decls tenv))))))

expand-decls : Sym × Listof(Decl) × Tenv → Listof(Decl)
(define expand-decls
```

```
  (lambda (m-name decls internal-tenv)
    (if (null? decls) ()
      (cases declaration (car decls)
        (opaque-type-decl (t-name)
          (let ((expanded-type
                  (qualified-type m-name t-name)))
            (let ((new-env
                    (extend-tenv-with-type
                      t-name expanded-type internal-tenv)))
              (cons
                (transparent-type-decl t-name expanded-type)
                (expand-decls
                  m-name (cdr decls) new-env)))))
        (transparent-type-decl (t-name ty)
          (let ((expanded-type
                  (expand-type ty internal-tenv)))
            (let ((new-env
                    (extend-tenv-with-type
                      t-name expanded-type internal-tenv)))
              (cons
                (transparent-type-decl t-name expanded-type)
                (expand-decls
                  m-name (cdr decls) new-env)))))
        (val-decl (var-name ty)
          (let ((expanded-type
                  (expand-type ty internal-tenv)))
            (cons
              (val-decl var-name expanded-type)
              (expand-decls
                m-name (cdr decls) internal-tenv)))))))))
```

Figure 8.10: Checker for OPAQUE-TYPES, part 2

The procedure expand-decls loops through a set of declarations, creating a new type environment in which every type or variable name is bound to an expanded type. One complication is that declarations follow let* scoping: each declaration in a set of declarations is in scope in all the following declarations.

To see what this means, consider the module definition

```
module m1
 interface
  [opaque t
   transparent u = int
   transparent uu = (t -> u)
   % point A
   f : uu
   ...]
body
 [...]
```

In order to satisfy the invariant, m1 should be bound in the type environment to an interface containing the declarations

```
[transparent t = from m1 take t
 transparent u = int
 transparent uu = (from m1 take t -> int)
 f : (from m1 take t -> int)
 ...]
```

If we do this, then any time we retrieve a type from this type environment, we will get an expanded type, as desired.

At point A, immediately before the declaration of f, the type environment should bind

```
t  to from m1 take t
u  to int
uu to (from m1 take t -> int)
```

We call the type environment at points like point A above the *internal* type environment. This will be passed as an argument to expand-decls.

We can now write expand-decls. Like defns->decls, this procedure creates only transparent declarations, since its purpose is to create a data structure in which qualified types can be looked up.

Last, we modify <:-decls to handle the two new kinds of declarations. We must now deal with the scoping relations inside a set of declarations. For example, if we are comparing

```
[transparent t = int
 x : bool             <: [y : int]
 y : t]
```

when we get to the declaration of y, we need to know that t refers to the type int. So when we recur down the list of declarations, we need to extend the type environment as we go, much as we built internal-tenv in expand-decls. We do this by calling extend-tenv-with-decl, which takes a declaration and translates it to an appropriate extension of the type environment (figure 8.11).

---

```
<:-decls : Listof(Decl) × Listof(Decl) × Tenv → Bool
(define <:-decls
  (lambda (decls1 decls2 tenv)
    (cond
      ((null? decls2) #t)
      ((null? decls1) #f)
      (else
        (let ((name1 (decl->name (car decls1)))
              (name2 (decl->name (car decls2))))
          (if (eqv? name1 name2)
            (and
              (<:-decl
                (car decls1) (car decls2) tenv)
              (<:-decls
                (cdr decls1) (cdr decls2)
                (extend-tenv-with-decl
                  (car decls1) tenv)))
            (<:-decls
              (cdr decls1) decls2
              (extend-tenv-with-decl
                (car decls1) tenv))))))))

extend-tenv-with-decl : Decl × Tenv → Tenv
(define extend-tenv-with-decl
  (lambda (decl tenv)
    (cases declaration decl
      (val-decl (name ty) tenv)
      (transparent-type-decl (name ty)
        (extend-tenv-with-type
          name
          (expand-type ty tenv)
          tenv))
      (opaque-type-decl (name)
        (extend-tenv-with-type
          name
          (qualified-type (fresh-module-name '%unknown) name)
          tenv)))))
```

---

Figure 8.11: Checker for OPAQUE-TYPES, part 3

We always use decls1 for this extension. To see why, consider the comparison

```
[transparent t = int          [opaque t
 transparent u = (t -> t) <:   transparent u = (t -> int)
 f : (t -> u)]                 f : (t -> (int -> int))]
```

This comparison should succeed, since a module body that supplies the bindings on the left would be a correct implementation of the interface on the right.

When we compare the two definitions of the type u, we need to know that the type t is in fact int. The same technique works even when the declaration on the left is not present on the right, as illustrated by the declaration of t in the first example above. We call expand-type to maintain the invariant that all types in the type environment are expanded. The choice of module names in the last clause of extend-tenv-with-decl doesn't matter, since the only operation on qualified types is equal?. So using fresh-module-name is enough to guarantee that this qualified type is new.

Now we get to the key question: how do we compare declarations? Declarations can match only if they declare the same name (either a variable or a type). If a pair of declarations have the same name, there are exactly four ways in which they can match:

- They are both value declarations, and their types match.
- They are both opaque type declarations.
- They are both transparent type declarations, and their definitions match.
- decl1 is a transparent type declaration, and decl2 is an opaque type declaration. For example, imagine that our module has an interface that declares opaque t and a body that defines type t = int. This should be accepted. The procedure defns-to-decls turns the definition type t = int into a transparent type declaration, so the test

  ```
  actual-iface <: expected-iface
  ```

  in add-module-defn-to-tenv will ask whether

  ```
  (transparent t = int) <:(opaque t)
  ```

  Since the module should be accepted, this test should return true.

  This tells us that something with a known type is always usable as a thing with an unknown type. But the reverse is false. For example,

  ```
  (opaque t) <:(transparent t = ty)
  ```

  should be false, because the value with an opaque type may have some actual type other than int, and a module that satisfies opaque t may not satisfy transparent t = int.

This gives us the code in figure 8.12. The definition of equiv-type? expands its types, so that in examples like

```
[transparent t = int x : bool y : t] <: [y : int]
```

above, the t on the left will be expanded to int, and the match will succeed.

---

```
<:-decl : Decl × Decl × Tenv → Bool
(define <:-decl
  (lambda (decl1 decl2 tenv)
    (or
      (and
        (val-decl? decl1)
        (val-decl? decl2)
        (equiv-type?
          (decl->type decl1)
          (decl->type decl2) tenv))
      (and
        (transparent-type-decl? decl1)
        (transparent-type-decl? decl2)
        (equiv-type?
          (decl->type decl1)
          (decl->type decl2) tenv))
      (and
        (transparent-type-decl? decl1)
        (opaque-type-decl? decl2))
      (and
        (opaque-type-decl? decl1)
        (opaque-type-decl? decl2)))))

equiv-type? : Type × Type × Tenv → Bool
(define equiv-type?
  (lambda (ty1 ty2 tenv)
    (equal?
      (expand-type ty1 tenv)
      (expand-type ty2 tenv))))
```

---

Figure 8.12: Checker for OPAQUE-TYPES, part 4

Exercise 8.16 [*] Extend the system of this section to use the language of exercise 7.24, and then rewrite exercise 8.15 to use multiple arguments instead of procedure-returning procedures.

Exercise 8.17 [**] As you did in exercise 8.8, remove the restriction that a module must produce the values in the same order as the interface. Remember, however, that the definition must respect scoping rules, especially for types.

Exercise 8.18 [**] Our code depends on the invariant that every type in a type environment is already expanded. We enforce this invariant by calling expand-type in many places in the code. On the other hand, it would be easy to break the system by forgetting to call expand-type. Refactor the code so that there are fewer calls to expand-type, and the invariant is maintained more robustly.

## 8.3 Module Procedures

The programs in OPAQUE-TYPES have a fixed set of dependencies. Perhaps module m4 depends on m3 and m2, which depends on m1. Sometimes we say the dependencies are *hard-coded*. In general, such hard-coded dependencies lead to bad program design, because they make it difficult to reuse modules. In this section, we add to our system a facility for *module procedures*, sometimes called *parameterized modules*, that allow module reuse. We call the new language PROC-MODULES.

### 8.3.1 Examples

Consider our three developers again. Charlie wants to use some of the facilities of Alice's module. But Alice's module uses a database that is supplied by Bob's module, and Charlie wants to use a different database, which is supplied by some other module (written by Diana).

To make this possible, Alice rewrites her code using module procedures. A module procedure is much like a procedure, except that it works with modules, rather than with expressed values. At the module level, interfaces are like types. Just as the type of a procedure in CHECKED specifies the type of its argument and the type of its result, the interface of a module procedure specifies the interface of its argument and the interface of its result.

Alice writes a new module Alices-point-builder that begins

```
module Alices-point-builder
 interface
  ((database : [opaque db-type
                opaque node-type
                insert-node : (node-type ->
                               (db-type -> db-type))
                ...])
  => [opaque point
      initial-point : (int -> point)
      ...])
```

This interface says that Alices-point-builder will be a module procedure. It will expect as an argument a module that will export two types, db-type and node-type, a procedure insert-node, and perhaps some other values. Given such a module, Alices-point-builder should produce a module that exports an opaque type point, a procedure initial-point, and perhaps some other values. The interface of Alices-point-builder also specifies a local name for its argument; we will see later why this is necessary.

The body of Alice's new module begins

```
body
 module-proc (m : [opaque db-type
                   opaque node-type
                   insert-node : (node-type ->
                                  (db-type -> db-type))
                   ...])
  [type point = ...
   initial-point = ... from m take insert-node ...
   ...]
```

Just as an ordinary procedure expression looks like

```
proc (var : t) e
```

a module procedure looks like

```
module-proc (m : [...]) [...]
```

In this example Alice has chosen m as the name of the bound variable in the module procedure; this need not be the same as the local name in the interface. We repeat the interface of the argument because the scope of a module interface never extends into the module body. This can be fixed (see exercise 8.27).

Now Alice rebuilds her module by writing

```
module Alices-points
 interface
  [opaque point
   initial-point : (int -> point)
   ...]
 body
  (Alices-point-builder Bobs-db-Module)
```

and Charlie builds his module by writing

```
module Charlies-points
 interface
  [opaque point
   initial-point : (int -> point)
   ...]
 body
  (Alices-point-builder Dianas-db-module)
```

Module Alices-points uses Bobs-db-module for the database. Module Charlies-points uses Dianas-db-module for the database. This organization allows the code in Alices-point-builder to be used twice. Not only does this avoid having to write the code twice, but if the code needs to be changed, the changes can be made in one place and they will be propagated automatically to both Alices-points and Charlies-points.

For another example, consider examples 8.11 and 8.12. In these two examples, we used what was essentially the same code for to-int. In example 8.11 it was

```
letrec int to-int (x : from ints1 take t)
            = if (z? x)
              then 0
              else -((to-int (p x)), -1)
```

and in example 8.12 the type of x was from ints2 take t. So we rewrite this as a module parameterized on the module that produces the integers in question.

Example 8.14: The declaration

```
module to-int-maker
 interface
  ((ints : [opaque t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)])
    => [to-int : (from ints take t -> int)])
 body
  module-proc (ints : [opaque t
                       zero : t
                       succ:(t->t)
                       pred:(t->t)
                       is-zero : (t -> bool)])
   [to-int
     = let z? = from ints take is-zero
       in let p = from ints take pred
       in letrec int to-int (x : from ints take t)
                    = if (z? x)
                      then 0
                      else -((to-int (p x)), -1)
       in to-int]
```

defines a module procedure. The interface says that this module takes as a module ints that implements the interface of arithmetic, and produces another module that exports a to-int procedure that converts ints's type t to an integer. The resulting to-int procedure cannot depend on the implementation of arithmetic, since here we don't know what that implementation is! In this code ints is declared twice: once in the interface and once in the body. This is because, as we said earlier, the scope of the declaration in the interface is local to the interface, and does not include the body of the module.

Let's look at a few examples of to-int in action:

Example 8.15

```
module to-int-maker ...as before...
```

```
module ints1 ...as before...

module ints1-to-int
 interface [to-int : (from ints1 take t -> int)]
 body
  (to-int-maker ints1)

let two1 = (from ints1 take succ
            (from ints1 take succ
             from ints1 take zero))
in (from ints1-to-int take to-int
    two1)
```

---

has type int and value 2. Here we first define the modules to-int-maker, and ints1. Then we apply to-int-maker to ints1, getting the module ints1-to-int, which exports a binding for from ints1-to-int take to-int.

Here's an example of to-int-maker used twice, for two different implementations of arithmetic.

has type int and value 0. If we had replaced (to-ints2 two2) by (to-ints2 two1), the program would not be well-typed, because to-ints2 expects an argument from the ints2 representation of arithmetic, and two1 is a value from the ints1 representation of arithmetic.

Exercise 8.19 [*] The code for creating two1 and two2 in example 8.16 is repetitive and therefore ready for abstraction. Complete the definition of a module

```
module from-int-maker
 interface
  ((ints : [opaque t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)])
    => [from-int : (int -> from ints take t)])
 body
  ...
```

that converts an integer expressed value to its representation in the module ints. Use your module to reproduce the computation of example 8.16. Use an argument bigger than two.

Example 8.16

---

```
module to-int-maker ...as before...

module ints1 ...as before...

module ints2 ...as before...

module ints1-to-int
 interface [to-int : (from ints1 take t -> int)]
 body (to-int-maker ints1)

module ints2-to-int
 interface [to-int : (from ints2 take t -> int)]
 body (to-int-maker ints2)

let s1 = from ints1 take succ
in let z1 = from ints1 take zero
in let to-ints1 = from ints1-to-int take to-int
in let s2 = from ints2 take succ
in let z2 = from ints2 take zero
in let to-ints2 = from ints2-to-int take to-int

in let two1 = (s1 (s1 z1))
in let two2 = (s2 (s2 z2))
in -((to-ints1 two1), (to-ints2 two2))
```

---

Exercise 8.20 [*] Complete the definition of the module

```
module sum-prod-maker
 interface
  ((ints : [opaque t
```

```
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)])
   => [plus : (from ints take t
               -> (from ints take t
                   -> from ints take t))
       times : (from ints take t
                -> (from ints take t
                    -> from ints take t))])
 body
  [plus = ...
   times = ...]
```

to define a module procedure that takes an implementation of arithmetic and produces sum and product procedures for that implementation. Use the definition of plus from page 33, and something similar for times.

Exercise 8.21 [*] Write a module procedure that takes an implementation of arithmetic ints and produces another implementation of arithmetic in which the number *k* is represented by the representation of 2 * *k* in ints.

Exercise 8.22 [*] Complete the definition of the module

```
module equality-maker
 interface
  ((ints : [opaque t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)])
   => [equal : (from ints take t
                -> (from ints take t
                    -> bool))])
 body
  ...
```

to define a module procedure that takes an implementation of arithmetic and produces an equality procedure for that implementation.

Exercise 8.23 [*] Write a module table-of that is similar to the tables module of exercise 8.15, except that it is parameterized over its contents, so one could write

```
module mybool-tables
 interface
  [opaque table
   empty : table
   add-to-table : (int ->
                   (from mybool take t ->
                    (table -> table)))
   lookup-in-table : (int ->
                      (table ->
                       from mybool take t))]
 body
  (table-of mybool)
```

to define a table containing values of type from mybool take t.

### 8.3.2 Implementation

### Syntax

Adding module procedures to our language is much like adding procedures. A module procedure has an interface that is much like a proc type.

*Iface* ::= ( (*Identifier* : *Iface*) => *Iface*)

| proc-iface (param-name param-iface result-iface) |
|---|

Although this interface looks a little like an ordinary procedure type, it is different in two ways. First, it describes functions from module values to module values, rather than from expressed values to expressed values. Second, unlike a procedure type, it

gives a name to the input to the function. This is necessary because the interface of the output may depend on the value of the input, as in the type of to-int-maker:

```
((ints : [opaque t
          zero : t
          succ : (t -> t)
          pred : (t -> t)
          is-zero : (t -> bool)])
 => [to-int : (from ints take t -> int)])
```

to-int-maker takes a module ints and produces a module whose type depends not just on the type of ints, which is fixed, but on ints it self. When we apply to-int-maker to ints1, as we did in example 8.16, we get a module with interface

```
[to-int : (from ints1 take t -> int)]
```

but when we apply it to ints2, we get a module with a different interface

```
[to-int : (from ints2 take t -> int)]
```

We extend expand-iface to treat these new interfaces as already expanded. This works because the parameter and result interfaces will be expanded when needed.

```
expand-iface : Sym × Iface × Tenv → Iface
(define expand-iface
  (lambda (m-name iface tenv)
    (cases interface iface
      (simple-iface (decls) ...as before...)
      (proc-iface (param-name param-iface result-iface)
        iface))))
```

We will need new kinds of module bodies to create a module procedure, to refer to the bound variable of a module procedure, and to apply such a procedure.

*ModuleBody* ::= `module-proc` (*Identifier* : *Iface*) *ModuleBody*

```
proc-module-body (m-name m-type m-body)
```

*ModuleBody* ::= *Identifier*

```
var-module-body (m-name)
```

*ModuleBody* ::= (*Identifier* *Identifier*)

```
app-module-body (rator rand)
```

## The Interpreter

We first add a new kind of module, analogous to a procedure.

```
(define-datatype typed-module typed-module?
  (simple-module
    (bindings environment?))
  (proc-module
    (b-var symbol?)
    (body module-body?)
    (saved-env environment?)))
```

We extend value-of-module-body to handle the new possibilities for a module body. The code is much like that for variable references and procedure calls in expressions (figure 8.13).

---

```
value-of-module-body : ModuleBody × Env → TypedModule
(define value-of-module-body
  (lambda (m-body env)
    (cases module-body m-body
```

```
(defns-module-body (defns) ...as before...)
(var-module-body (m-name)
  (lookup-module-name-in-env m-name env))
(proc-module-body (m-name m-type m-body)
  (proc-module m-name m-body env))
(app-module-body (rator rand)
  (let ((rator-val
          (lookup-module-name-in-env rator env))
        (rand-val
          (lookup-module-name-in-env rand env)))
    (cases typed-module rator-val
      (proc-module (m-name m-body env)
        (value-of-module-body m-body
          (extend-env-with-module
            m-name rand-val env)))
      (else
        (report-bad-module-app rator-val))))))))
```

Figure 8.13: value-of-module-body

### The Checker

We can write down rules like the ones in section 7.2 for our new kinds of module bodies. These rules are shown in figure 8.14. We write (▷ *body tenv*)=*i* instead of (interface-of *body tenv*) = *i* in order to make the rules fit on the page.

$$\text{IFACE-M-VAR}$$
$$(\triangleright\ m\ \ tenv)\ =\ tenv(m)$$

$$\text{IFACE-M-PROC}$$
$$\frac{(\triangleright\ body\ \ [m{=}i_1]\,tenv)\ =\ i_1'}{(\triangleright\ (\texttt{m-proc}\ \ (m:i_1)\ \ body)\ \ tenv)\ =\ ((m:i_1)\ \texttt{=>}\ i_1')}$$

$$\text{IFACE-M-APP}$$
$$\frac{tenv(m_1) = ((m:i_1)\ \texttt{=>}\ i_1') \qquad tenv(m_2) = i_2 \qquad i_2 <: i_1}{(\triangleright\ (m_1\ \ m_2)\ \ tenv)\ =\ i_1'[m_2/m]}$$

Figure 8.14: Rules for typing new module bodies

A module variable gets its type from the type environment, as one might expect. A module-proc gets its type from the type of its parameter and the type of its body, just like the procedures in CHECKED.

An application of a module procedure is treated much like a procedure call in CHECKED. But there are two important differences.

First, the type of the operand ($i_2$ in the rule IFACE-M-APP) need not be exactly the same as the parameter type ($i_1$). We require only that $i_2 <: i_1$. This is sufficient, since $i_2 <: i_1$ implies that any module that satisfies the interface $i_2$ also satisfies the interface $i_1$, and is therefore an acceptable argument to the module procedure.

Second, we substitute the operand $m_2$ for $m$ in the result type $i_1'$. Consider the example on page 318, where we applied the module procedure to-int-maker, which has the interface

```
((ints : [opaque t
          zero : t
          succ : (t -> t)
          pred : (t -> t)
          is-zero : (t -> bool)])
  => [to-int : (from ints take t -> int)])
```

to ints1 and ints2. When we apply to-int-maker to ints1, the substitution gives us the interface

```
[to-int : (from ints1 take t -> int)]
```

When we apply it to ints2, the substitution gives the interfaces

```
[to-int : (from ints2 take t -> int)]
```

as desired.

From these rules, it is easy to write down the code for interface-of (figure 8.15). When we check the body of a module-proc, we add the parameter to the type environment as if it had been a top-level module. This code uses the procedure rename-in-iface to perform the substitution on the result interface.

---

```
interface-of : ModuleBody × Tenv → Iface
(define interface-of
  (lambda (m-body tenv)
    (cases module-body m-body
      (var-module-body (m-name)
        (lookup-module-name-in-tenv tenv m-name))
      (defns-module-body (defns)
        (simple-iface
          (defns-to-decls defns tenv)))
      (app-module-body (rator-id rand-id)
        (let ((rator-iface
                (lookup-module-name-in-tenv tenv rator-id))
              (rand-iface
                (lookup-module-name-in-tenv tenv rand-id)))
          (cases interface rator-iface
            (simple-iface (decls)
              (report-attempt-to-apply-simple-module rator-id))
            (proc-iface (param-name param-iface result-iface)
              (if (<:-iface rand-iface param-iface tenv)
                (rename-in-iface
                  result-iface param-name rand-id)
                (report-bad-module-application-error
                  param-iface rand-iface m-body))))))
      (proc-module-body (rand-name rand-iface m-body)
        (let ((body-iface
                (interface-of m-body
                  (extend-tenv-with-module rand-name
                    (expand-iface rand-name rand-iface tenv)
                    tenv))))
          (proc-iface rand-name rand-iface body-iface))))))
```

---

Figure 8.15: Checker for PROC-MODULES, part 1

Last, we extend <:-iface to handle the new types. The rule for comparing proc-ifaces is

$$\frac{i_2 <: i_1 \qquad i_1'[m'/m_1] <: i_2'[m'/m_2] \qquad m' \text{ not in } i_1' \text{ or } i_2'}{((m_1 : i_1) \Rightarrow i_1') <: ((m_2 : i_2) \Rightarrow i_2')}$$

In order to have $((m_1{:}i_1) \Rightarrow i_1')$ <: $((m_2{:}i_2) \Rightarrow i_2')$, it must be the case that any module $m_0$ that satisfies the first interface also satisfies the second interface. This means that any module with interface $i_2$ can be passed as an argument to $m_0$, and any module that $m_0$ produces will satisfy $i_2'$.

For the first requirement, we insist that $i_2$ <: $i_1$. This guarantees that any module that satisfies $i_2$ can be passed as an argument to $m_0$. Note the reversal: we say that subtyping is *contravariant* in the parameter type.

What about the result types? We might require that $i_1'$ <: $i_2'$. Unfortunately, this doesn't quite work. $i_1'$ may have instances of the module variable $m_1$ in it, and $i_2'$ may have instances of $m_2$ in it. So to compare them, we rename both $m_1$ and $m_2$ to some new module variable $m'$. Once we do that, we can compare them sensibly. This leads to the requirement $i_1'[m'/m_1]$ <: $i_2'[m'/m_2]$.

The code to decide this relation is relatively straightforward (figure 8.16). When deciding $i_1'[m'/m_1]$ <: $i_2'[m'/m_2]$ we extend the type environment to provide a binding for $m'$. We associate $m'$ with $i_1$, since it has fewer components than $i_2$. When we call extend-tenv-with-module to compare the result types, we call expand-iface to maintain the invariant.

```
<:-iface : Iface × Iface × Tenv → Bool
(define <:-iface
  (lambda (iface1 iface2 tenv)
    (cases interface iface1
      (simple-iface (decls1)
        (cases interface iface2
          (simple-iface (decls2)
            (<:-decls decls1 decls2 tenv))
          (proc-iface (param-name2 param-iface2 result-iface2)
            #f)))
      (proc-iface (param-name1 param-iface1 result-iface1)
        (cases interface iface2
          (simple-iface (decls2) #f)
          (proc-iface (param-name2 param-iface2 result-iface2)
            (let ((new-name (fresh-module-name param-name1)))
              (let ((result-iface1
                      (rename-in-iface
                        result-iface1 param-name1 new-name))
                    (result-iface2
                      (rename-in-iface
                        result-iface2 param-name2 new-name)))
                (and
                  (<:-iface param-iface2 param-iface1 tenv)
                  (<:-iface result-iface1 result-iface2
                    (extend-tenv-with-module
                      new-name
                      (expand-iface new-name param-iface1 tenv)
                      tenv)))))))))))
```

Figure 8.16: Checker for PROC-MODULES, part 2

And now we're done. Go have a sundae, with anything that satisfies the ice cream interface, anything that satisfies the hot-topping interface, and anything that satisfies the nuts interface. Don't worry about how any of the pieces are constructed, so long as they taste good!

Exercise 8.24 [*] Application of modules is currently allowed only for identifiers. What goes wrong with the type rule for application if we try to check an application like (m1 (m2 m3))?

Exercise 8.25 [*] Extend PROC-MODULES so that a module can take multiple arguments, analogously to exercise 3.21.

Exercise 8.26 [**] Extend the language of module bodies to replace the production for module application by

$$\mathit{ModuleBody} ::= (\mathit{ModuleBody}\ \ \mathit{ModuleBody})$$

```
app-module-body (rator rand)
```

Exercise 8.27 [***] In PROC-MODULES, we wind up having to write interfaces like

```
[opaque t
 zero : t
 succ : (t -> t)
 pred : (t -> t)
 is-zero : (t -> bool)]
```

over and over again. Add to the grammar for programs a facility for named interfaces, so we could write

```
interface int-interface = [opaque t
                           zero : t
```

```
                            succ : (t -> t)
                            pred : (t -> t)
                            is-zero : (t -> bool)]
 module make-to-int
  interface
   ((ints : int-interface)
    => [to-int : from ints take t -> int])
  body
   ...
```