CSG111 March 27, 2007
Dr. Wand Readings: EOPL3, Chapter 6

Lecture 7: Modules

Key Concepts:

Module
Interface
Abstraction barrier or boundary
`let*` scoping
Qualified Variable
Abstract Type
Type Abbreviation
Qualified Type
Parameterized module
Subtyping
    contravariant, covariant

7.1 Introduction

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 portions, and to document the dependencies between those portions.

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 becomes awkward to use when programs may be large or split up over many files.

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 easily 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.

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

Modules are connected using a *module interconnection language*. This language is largely independent of the base language, and is our primary focus in this chapter.

A program in the module interconnection language consists of a set of *module definitions*. Each definition binds a name to a *module value*. A module value is either a set of bindings, much like an environment, or a module procedure that takes a module value and produces another module value.

Each module will have a *module type*. A module type can be an *interface*, which is much like a type environment, or a procedure module type, which are analogous to ordinary procedure types, except that they describe the argument and result of a module procedure.

We would like to be able to check statically whether our programs respect their abstraction boundaries. We therefore emphasize the types of our example programs, since their values are straightforward to compute. 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.

7.2 The Basic Module System

Example 1. The program

```
module m1
 interface [u : int]
 implementation [u = 33]

import m1
from m1 take u
```

has type `int` and value 33.

The first three lines are a definition of a module named `m`. It has an *interface* `[u : int]` and a *implementation* `[u = 33]`. We say that `u` is *declared* in the interface and *defined* in the module implementation. Alternatively, we say that the module `m1` *exports* a value for `u`.

The last two lines constitute the *program body*, which imports the module `m1` and evaluates the expression `from m1 take u`. We adopt the convention that we separate module definitions from the program body by a blank line, but that is not required by our grammar.

The clause `import m1` is like a `let`. It causes the value of the module `m1` to be computed. If there were effects in the definition of `u`, this is when they would occur.

The expression `from m take u` is called a *qualified variable reference*. In conventional languages it might be written `m.u` or `m:u` or `m::u`.

In the program body, `m` is bound to a module value that associates the name `u` with the expressed value 3. The qualified variable reference `from m take u` denotes the binding of the name `u` in module `m`. Hence its type is `int` and its value is 33.

Example 2. The interface establishes an *abstraction barrier* between the module and the program body. We sometimes say that the expressions in the implementation are *inside* the abstraction barrier, and everything else is *outside* the abstraction barrier. A module body may supply bindings for names that are not in the interface, but those bindings are not visible in the program body.

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

import m1
from m1 take u
```

has type `int` and value 33, but

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

import m1
from m1 take v
```

has no type, since the body of the program is checked against the interface of `m1`, without crossing the abstraction boundary.

Example 3. The program

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

44
```

has no type. 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 4. A module can produce a module value containing more than one binding. For example, the program

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

import m1
-(from m take u, from m take v)
```

has type `int`.

Example 5. The module body must supply bindings for all the declarations in the interface. For example,

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

import m1
from m1 take u
```

has no type, because the implementation of `m1` does not provide all of the values that its interface advertises.

Example 6. To keep the implementation simple, our language requires that the module produce the values in the same order as the interface. Hence

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

from m1 take u
```

has no type. This can be fixed.

Example 7. In our language, the body of the module has `let*` scoping, so in the following program, the definition of `u` is in scope in the rest of the definitions.

```
module m1
 interface
  [u : int
   v : int]
 implementation
  [u = 44
   v = -(u,33)]

import m
-(from m take u, from m take v)
```

has type `int`.

Example 8. In our language, module names are global. However, the values exported by a module are computed only when that module is imported. In order to use a value from a module, that module must be explicitly imported. This is required even inside another module.

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

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

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

has type `int`, as does

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

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

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

If we had omitted any of the `import` clauses, these programs would both be ill-typed.

Example 9. If a module tries to import itself, even indirectly, then an infinite loop will result. For example,

```
module m1
 interface [v : int]
 implementation
  import m2
  [v = 11]

module m2
 interface [v : int]
 implementation
  import m1
  [v = 12]

import m1
44
```

will cause an infinite loop, since the attempt to import `m1` will attempt to import `m2`, which will attempt to import `m1` again. In this program, of course, the imports are completely unnecessary.

7.3 Implementing the Basic Module System

7.3.1 Syntax

A program consists of a sequence of module definitions, followed by an expression.

```
(program
  ((arbno module-definition)
   (arbno "import" identifier)
   expression)
  a-program)
```

A module definition consists of its name, its type, and its implementation, which consists of an arbitrary number of import statements and a module body.

```
(module-definition
  ("module" identifier
    "interface" module-type
    "implementation"
     (arbno "import" identifier)
     module-body)
  a-module-definition)
```

A module type 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 more declarations and more module types.

```
(module-type
  ("[" (arbno declaration) "]")
  env-module-type)

(declaration
  (identifier ":" type)
  val-declaration)
```

A module value has an `env-module-type` if and only if it is an environment in which each variable is bound to a value of the right type.

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

```
(module-body
  ("[" (arbno definition) "]")
  defns-module-body)

(definition
  (identifier "=" expression)
  val-definition)
```

Last, we have one new expression, a qualified variable reference.

```
(expression
  ("from" identifier "take" identifier)
  qualified-var-ref)
```

7.3.2 The Interpreter

The value of a module will be an environment. To allow a module `m` to be bound to a module value, we extend our notion of environments to allow bindings to modules.

```
(define-datatype module-value module-value?
  (env-module-value
    (bindings environment?)))
```

Later, we will add another variant to handle the values of module procedures.

```
(define-datatype environment environment?
  (empty-env)
  (extend-env ...)
  (extend-env-recursively ...)
  (extend-env-with-module
    (m-name symbol?)
    (m-value module-value?)
    (saved-env environment?)))
```

We modify `apply-env` to look only at `extend-env` and `extend-env-recursively` bindings, and we add a procedure `lookup-module-in-env` to look up bindings made by `extend-env-with-module`. In general, we will use the name `lookup-`X`-in-`Y for a procedure that looks up things of kind X in data structures of kind Y.

To evaluate a qualified reference, we first retrieve the module from the environment, and then we look up the variable in the environment exported by the module.

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (qualified-var-ref (m-name var-name)
        (let ((m-val (lookup-module-in-env m-name env)))
          (cases module-value m-val
            (env-module-value (env)
              (apply-env env var-name)))))
      ...)))
```

To evaluate a program, we evaluate its body in an initial environment built by importing all the modules in its import specification.

```
;; value-of-program : program -> expval

(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (modules imports body)
        (let ((env (initial-env imports modules)))
          (value-of body env)))
      )))

;; (list-of sym) * (list-of module-definition) -> environment
(define initial-env
  (lambda (imports modules)
    (if (null? imports)
      (empty-env)
      (extend-env-with-module
        (car imports)
        (value-of-module
          (lookup-module-in-list (car imports) modules)
          modules)
        (initial-env (cdr imports) modules)))))
```

To import a module, we evaluate its body in an environment constructed by importing all of its imports.

```
;; module-definition * module-list -> module-value
(define value-of-module
  (lambda (m-def modules)
    (cases module-definition m-def
      (a-module-definition (name m-type imports m-body)
        (value-of-module-body m-body
          (initial-env imports modules))))))
```

Last, to evaluate a module body, we build an environment, evaluating each expression in the appropriate environment to get `let*` scoping.

```
;; value-of-module-body : module-body * env -> module-value
(define value-of-module-body
  (lambda (m-body env)
    (cases module-body m-body
      (defns-module-body (defns)
        (env-module-value
          (defns-to-env defns env))))))

;; defns-to-env : (list-of definition) * env -> env

(define defns-to-env
  (lambda (defns env)
    (if (null? defns)
      env
      (cases definition (car defns)
        (val-definition (var exp)
          (let ((val (value-of exp env)))
            (defns-to-env
              (cdr defns)
              (extend-env var val env))))))))
```

7.3.3 The Checker

So far, our language is just a more complicated version of `letrec`. So we check a program in our language much as we checked a `letrec`: first we assemble the declared types of each module, then we check each module against its declared type, and last we find the type of the program body.

If we think of each module as producing a data type, then the declared types are the interface of each module, and the body of the module is the implementation of that interface. The interfaces are public, but the implementations are private. When we check any piece of code, we check it only against the public data: the interfaces. When checking the code of a module, we do not look at the implementation of any other module. When checking the body of the program, we do not look at the implementation of any module at all.

As we did with the interpreter, our first step is to allow module bindings in the type environment.

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

The overall process of checking a program has three steps.

- The first step is to create a *manifest* consisting of the name and declared type of each module in the program.
- In the second step, we check each module definition to see that it produces a module value consistent with its declared type. Any cross-module references are resolved using the manifest, without looking inside the implementation of the other module. Thus the manifest implements the abstraction boundary around each module.
- Finally, we find the type of the body of the program.

```
;; type-of-program : program -> expanded-type
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (modules imports body)
        ((the-manifest
           (module-definitions->manifest modules)))
        (for-each
          (lambda (m-defn)
            (check-module-definition!
              m-defn the-manifest))
          modules)
        (let ((tenv (initial-tenv imports the-manifest)))
          (expand-answer
            (type-of body tenv (empty-subst)))))))))
```

Our expressions will be written in INFERRED, so we use the type inference machinery of the last lecture. In this system, `type-of` returns a type and a substitution in which to interpret its type variables. We use `expand-answer` to eliminate these type variables.

```
(define expand-answer
  (lambda (ans)
    (cases answer ans
      (an-answer (ty subst)
        (apply-subst-to-type ty subst)))))
```

To check a module definition, we compute the type of its body in a type environment consisting of the types of each of its imports. We then compare the actual type to the expected type using the procedure `<:-module-type`.

Like `check-equal-type!`, the procedure `check-module-definition!` is executed for effect only.

```
;; m-defn * manifest -> unspecified
(define check-module-definition!
  (lambda (m-defn manifest)
    (cases module-definition m-defn
      (a-module-definition (m-name expected-type imports m-body)
        (let ((tenv-for-m-body
                (initial-tenv imports manifest)))
          (let ((actual-type
                  (type-of-module-body m-body tenv-for-m-body)))
            (if (not (<:-module-type
                       actual-type
                       expected-type
                       manifest))
              (raise-bad-module-defn-error m-name
                expected-type actual-type))))))))
```

To find the type of a module body, we must create a module type that associates each variable defined in the body with the type of its definition. Recall that a module type contains a list of declarations. The procedure `defns-to-decls` creates such a list. At every step it also extends the local type environment, to follow the correct `let*` scoping. Each expression is evaluated in an empty substitution, so type inference is possible only within a expression. We call `expand-answer`, as we did above, to expand the type variables in the substitution returned by `type-of`.

```
;; module-body * tenv -> module-type
(define type-of-module-body
  (lambda (m-body tenv)
    (cases module-body m-body
      (defns-module-body (defns)
        (env-module-type
          (defns-to-decls defns tenv))))))

;; defns-to-decls : defns * tenv -> decls
(define defns-to-decls
  (lambda (defns tenv)
    (if (null? defns) '()
      (cases definition (car defns)
        (val-definition (name exp)
          (let ((ty (expand-answer
                      (type-of exp tenv (empty-subst)))))
            (cons
              (val-declaration name ty)
              (defns-to-decls (cdr defns)
                (extend-tenv name ty tenv)))))))))
```

All that's left is to compare the actual and expected types of each module, using the procedure `<:-module-type`. We think of `<:` as "subtype". We will define subtyping so that if type t_1 is a subtype of a type t_2 then every value of type t_1 can be used as a value of t_2. This principle will apply not just to ordinary types, but to module types as well. For example

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

since any module value satisfying 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, `<:-module-type` just calls `<:-declarations`, which compares declarations. These procedures take a `tenv` argument that is not used for the basic module system, but will be needed in section 4.

```
(define <:-module-type
  (lambda (m-type1 m-type2 tenv)
    (cases module-type m-type1
      (interface-module-type (decls1)
        (cases module-type m-type2
          (interface-module-type (decls2)
            (<:-declarations decls1 decls2 tenv)))))))
```

The procedure `<:-declarations` does the main work of comparing two sets of declarations. If `decls1` and `decls2` are two sets of declarations, we say `decls1 <: decls2` if and only if any environment that supplies bindings for the declarations of `decls1` also supplies bindings for the declarations for `decls2`. This can be assured if `decls1` contains a matching declaration for every declaration in `decls2`, as in the example above.

The procedure `<:-declarations` 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`.

```
(define <:-declarations
  (lambda (decls1 decls2 tenv)
    (cond
      ((null? decls2) #t)
      ((null? decls1) #f)
      (else
        (cases declaration (car decls2)
          (val-declaration (name2 ty2)
            (cases declaration (car decls1)
              (val-declaration (name1 ty1)
                (if (eqv? name1 name2)
                  (and
                    (<:-type ty1 ty2 tenv)
                    (<:-declarations
                      (cdr decls1) (cdr decls2) tenv))
                  (<:-declarations
                    (cdr decls1) decls2 tenv))))))))))
```

Last, we must write `<:-type`. If we did not have type inference in our language, this would be a simple equality test. However, since we do have type inference, we must deal with the possibility that we may have type variables in our types. For example, in

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

the actual type for `f` reported by the type inference engine will be something like `(tvar07 -> tvar07)`, and we will need to conclude that `(tvar07 -> tvar07) <: (int -> int)`. This is correct, since the polymorphic function can be used as an `(int -> int)`. On the other hand we should reject the assertion `(tvar07 -> tvar07) <: (int -> bool)`. To accomplish this task, `<:-type` performs a recursive traversal of the two types, much like the unifier does. As it walks through the types, it collects information about the substitution of each type variable to make sure that type variables are substituted consistently. This will allow it to accept

```
(tvar07 -> tvar07) <: (int -> int)
```

but reject

```
(tvar07 -> tvar07) <: (int -> bool)
```

as it should. We write `(maybe subst)` for the result type of `<:-type` to indicate that it either returns a substitution or `#f`. We wrap the first call to `recur` in a `(cond ((...)  => ...))` because it might return `#f`.

```
;; <:-type : type * type * tenv -> (maybe subst)

(define <:-type
  (lambda (ty1 ty2 tenv)
    (let recur
      ((ty1 ty1)
       (ty2 ty2)
       (subst (empty-subst)))
      (cond
        ((equal? ty1 ty2) subst)
        ((tvar-type? ty1)
         (extend-subst ty1 ty2 subst))
        ((tvar-type? ty2)
         (recur ty2 ty1 subst))
        ((and (proc-type? ty1) (proc-type? ty2))
         (cond
           ((recur
              (proc-type->arg-type ty1)
              (proc-type->arg-type ty2)
              subst)
            =>
            (lambda (subst1)
              (recur
                (proc-type->result-type ty1)
                (proc-type->result-type ty2)
                subst1)))
           (else #f)))
        (else #f)))))
```

7.4 Modules that declare types

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

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

declares a type `t`, and some operations `zero`, `succ`, `pred`, `is-zero` that operate on values of that type. This is the interface that might be associated with an implementation of arithmetic.

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

7.4.1 Type Abbreviations

We begin by discussing type abbreviations, or *transparent types*.

Example 10. The program

```
module m1
 interface [type-abbrev t = int
            z : t]
 implementation [type t = int
                 z = 0]

import m1
-(from m1 take z,1)
```

has type `int`. Here the declaration `type-abbrev t = int` in the interface binds `t` to the type `int` in the rest of the interface. The declaration `type t = int` in the module body, on the third line of the program, binds `t` to the type `int` in the rest of the module body. We say that `t` is *bound transparently* to `int`, or that `t` is a *transparent type*.

Example 11. Of course, we can use any name we like for the type.

```
module m1
 interface [type-abbrev u = int
            z : u]
 implementation [type u = int
                 z = 0]

import m1
-(from m1 take z,1)
```

has type `int`. And of course 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.

Example 12. If a module `m` exports a type named `t`, we can refer to that type wherever `m` has been imported by writing `from m take t`. We call this a *qualified type*.

If `t` is a transparent type, then `from m take t` is an abbreviation for the meaning of `t`.

```
module m1
 interface [type-abbrev t = int
            z : t]
 implementation [type t = int
                 z = 0]

import m1
proc (x : from m1 take t) -(x,1)
```

has type `(int -> int)`. Since `t` has been exported transparently, we can use `from m1 take t` outside the module definition as an abbreviation for `int`. Here we have used it to declare the type of the bound variable `x`.

Example 13. A type definition acts like a type abbreviation inside the module body.

```
module m1
 interface [type-abbrev t = int
            f : (t -> t)]
 implementation [type t = int
                 f = proc (x : t) -(x,1)]
import m1
(from m1 take f
 33)
```

has type `int`.

Example 14. In our language, types are public; they do not need to be imported. For example, the program

```
module m1
 interface [type-abbrev t = int
            z : t]
 implementation [type t = int
                 z = 0]
module m2
 interface
  [foo : (from m1 take t -> int)]
 implementation
  [foo = proc (x : from m1 take t) x]

import m2
from m2 take foo
```

has type `(int -> int)`, even though `m2` does not import `m1`

7.4.2 Abstract Type Declarations

A module can also export *abstract* or *opaque* types by using an `abstract-type` declaration.

Example 15. The program

```
module m1
 interface [type-abbrev t = int
            z : t
            check : (t -> bool)]
 implementation [type t = int
                 z = 0
                 check = proc (x : t) zero?(x)]

import m1
from m1 take z
```

has type `int`, but the program

```
module m1
  interface [abstract-type t
             z : t]
  implementation [type t = int
                  z = 0]

import m1
from m1 take z
```

has type `from m1 take t`. The declaration `abstract-type t` in the interface, on the first line of the program, declares `from m1 take t` to be an abstract type, meaning that outside the module, it is a new base type, like `int` or `bool`. The corresponding definition `type t = int` declares `t` to be an abbreviation for `int` inside the module body, but this information is hidden from the rest of the program. This is the abstraction boundary.

The body of the module is the implementation of the interface, and is said to be inside the abstraction boundary. The rest of the program is the client of the interface and is said to be outside the abstraction boundary.

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.

Example 16. The program

```
module m1
 interface [abstract-type t
            z : t
            check : (t -> bool) ]
 implementation [type t = int
                 z = 0
                 check = proc (x : t) zero?(x)]
import m1
-(from m1 take z, 1)
```

has no type. The program outside the abstraction boundary cannot rely on the fact that the value of `from m1 take z` is an integer.

Example 17. A module can export an abstract type and some operations on that abstract type. Then the only way that the rest of the program can operate on values of the abstract type is through these operations. For example, the program

```
module m1
 interface [abstract-type t
            z : t
            check : (t -> bool) ]
 implementation [type t = int
                 z = 0
                 check = proc (x : t) zero?(x)]
import m1
let f = from m1 take check
in let a = from m1 take z
   in (f a)
```

has type `bool`. The procedure `from m1 take check` has type `(from m1 take t -> bool)`. The value `from m1 take z` has type `from m1 take t`, so it is an acceptable argument to the procedure.

Example 18. Here is 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.

```
module mybool
 interface [abstract-type t
            true : t
            false : t
            and : (t -> (t -> t))
            not : (t -> t)
            to-bool : (t -> bool)]
 implementation [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)
                            if zero?(x)
                             then zero?(0)
                             else zero?(1)]

import mybool
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.

Example 19. Here is a more elaborate that implements a simple abstraction of tables. Our tables are like environments, except that instead of binding symbols to Scheme values, they binds 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 multi-argument procedures by using Currying.

```
module tables
 interface [abstract-type table
            empty : table
            add-to-table : (int ->
                            (int ->
                             (table -> table)))
            lookup-in-table : (int ->
                               (table -> int))]
 implementation
  [type table = (int -> int)
   type keytype = int
   type valtype = int
   empty = proc (n : int) 0
   add-to-table
    = proc (key : keytype)
       proc (val : valtype)
        proc (saved-table : table)
         proc (search-key : keytype)
          if zero?(-(key, search-key))
           then val
           else (saved-table search-key)
    lookup-in-table
     = proc (search-key : keytype)
        proc (table : table)
         (table search-key)]
```

```
import tables
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) empty)
in let table2 = (((add-binding 4) 400) table1)
in let table3 = (((add-binding 3) 301) table2)
in -( ((lookup 4) table3),
      ((lookup 3) table3))
```

This program has type `int`. The table `table3` binds 4 to 400 and 3 to 301, so the value of the program is 99.

7.4.3 Implementation

We now need to extend our system to model type abbreviations, abstract types, and qualified type references.

7.4.3.1 Syntax

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

```
(type
  (identifier)
  named-type)

(type
  ("from" identifier "take" identifier)
  qualified-type)
```

We add two new kinds of declarations, for type abbreviations and for abstract types.

```
(declaration
  ("abstract-type" identifier)
  abstract-type-declaration)

(declaration
  ("type-abbrev" identifier "=" type)
  type-abbrev-declaration)
```

We also add a new kind of definition: a type definition:

```
(definition
  ("type" identifier "=" type)
  type-definition)
```

7.4.3.2 The Interpreter

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

```
(define defns-to-env
  (lambda (defns env)
    (if (null? defns)
      (empty-env)
      (cases definition (car defns)
        (val-definition (var exp) ...)
        (type-definition (var ty)
          (defns-to-env (cdr defns) env))))))
```

7.4.3.3 The Checker

The changes to the checker are more substantial, since many portions of the checker deal with types, and they must all be extended to handle the new types.

First, we extend type environments to handle new types. We will have two new kinds of bindings: for abstract types and for type abbreviations.

```
(define-datatype type-environment type-environment?
  (empty-tenv)
  (extend-tenv ...)
  (extend-tenv-with-module ...)
  (extend-tenv-with-abstype
    (t-name symbol?)
    (saved-tenv type-environment?))
  (extend-tenv-with-type-abbrev
    (t-name symbol?)
    (t-type type?)
    (saved-tenv type-environment?))
  )
```

As we did in the previous checker, we will construct a global type environment in which each module is bound to its type.

Next, we introduce a systematic way of handling abstract types and type abbbre-viations. We have said that an abstract type behaves like a primitive type, such as `int` or `bool`. Type abbreviations, on the other hand, are transparent: they behave exactly like their expansions. So every type is equivalent to one that is given by the grammar

$$\begin{aligned}
\textit{Type} &::= \texttt{int} \\
\textit{Type} &::= \texttt{bool} \\
\textit{Type} &::= t_1 \\
\textit{Type} &::= \texttt{(}\textit{Type}\ \texttt{->}\ \textit{Type}\texttt{)} \\
\textit{Type} &::= \texttt{from}\ m\ \texttt{take}\ t_2
\end{aligned}$$

where t_1 is bound as an abstract type in the current type environment, or where t_2 is declared as an abstract type in *m*. We call a type of this form an *expanded type*.

We write a procedure, `expand-type-in-tenv`, that takes a type and expands it to get a type of this form. For example, if `tenv` contains a declaration for a module

```
module m1
 interface
  [abstract-type t
   type-abbrev u = int
   type-abbrev uu = (t -> u)]
 ...
```

then calling `expand-type-in-tenv` on `from m1 take u` should return `int`, and calling it on `from m1 take uu` should return

```
(from m1 take t -> int)
```

since `from m1 take t` is an abstract type.

We perform this expansion by examining the type. If the type `ty` is a `proc` type, we recur on the type of the argument and the type of the result. If the type is a named type, then we call `type-abbrev->maybe-binding-in-tenv` to see if it is bound transparently. If it is, then we recur on its definition. If it is a qualified type, we call `expand-qualified-type-in-tenv` to handle the job. If none of these cases apply, then the type must be `int` or `bool` or a named type that is bound as an abstract type in the type environment. In any of these situations, the type is already expanded.

```
(define expand-type-in-tenv
  (lambda (ty tenv)
    (cases type ty
      (proc-type (arg-type result-type)
        (proc-type
          (expand-type-in-tenv arg-type tenv)
          (expand-type-in-tenv result-type tenv)))
      (named-type (name)
        (cond
          ((type-abbrev->maybe-binding-in-tenv name tenv)
           => (lambda (ty1) (expand-type-in-tenv ty1 tenv)))
          (else ty)))
      (qualified-type (m-name t-name)
        (expand-qualified-type-in-tenv m-name t-name tenv))
      (else ty)   ;; this covers int, bool, and tvar-types.
      )))
```

To expand a qualified type `from` m_1 `take` t_1, first we find the type of m_1 in the current type environment. If the type of m_1 declares t_1 as a type abbreviation, its definition is returned, with m_1 wrapped around any named types in the definition. For example, if `m1` is bound as in the example above, applying this process to `from m1 take uu` will get

```
(from m1 take t -> from m1 take u)
```

This type is not completely expanded, of course, so we then call `expand-type-in-tenv` to finish the job, expanding it to

```
(from m1 take t -> int)
```

as desired.

```
  (define expand-qualified-type-in-tenv
    (lambda (m-name t-name tenv)
      (let ((ty (qualified-type m-name t-name)))
        (cond
          ((module-name->maybe-binding-in-tenv tenv m-name)
           => (lambda (m-type)
                (cases module-type m-type
                  (env-module-type (decls)
                    (cond
                      ((type-abbrev->maybe-binding-in-decls t-name decls)
                       =>
                       ;; yes, wrap any named types in
                       ;; qualified-type, and then continue to expand
                       (lambda (ty)
                         (expand-type-in-tenv
                           (wrap-named-types-with-m-name ty m-name)
                           tenv)))
                      (else ty))))))
          (else ty)))))
```

The procedure `wrap-named-types-with-m-name` recurs on the type, wrapping any named types in the module name.

```
(define wrap-named-types-with-m-name
  (lambda (ty m-name)
    (let recur ((ty ty))
      (cases type ty
        (int-type () ty)
        (bool-type () ty)
        (proc-type (arg-type result-type)
          (proc-type (recur arg-type) (recur result-type)))
        (named-type (ty-name) (qualified-type m-name ty-name))
        (qualified-type (modname varname) ty)
        (tvar-type (serial-number)
          (eopl:error 'wrap-named-types-with-m-name
            "can't expand a tvar type ~s"
            ty))
          ))))
```

We can now go through the checker, making modifications to handle the new types.

For our new checker, we make sure that all types in the manifest are already expanded, so we don't have to call `expand-type-in-tenv` every time we see a type. To do this, we build the manifest in two stages. First we build a type environment consisting of all the module definitions. Then we build a second type environment just like the first, except that all the types are expanded. Here the procedure `expand-module-type-in-tenv` walks through a module type, expanding all the types it finds.

```
(define module-definitions->manifest
  (lambda (defns)
    (build-expanded-manifest defns
      (build-primitive-manifest defns))))

(define build-primitive-manifest
  (lambda (defns)
    (if (null? defns)
      (empty-tenv)
      (extend-tenv-with-module
        (module-definition->name (car defns))
        (module-definition->module-type (car defns))
        (build-primitive-manifest (cdr defns))))))

(define build-expanded-manifest
  (lambda (defns tenv)
    (let recur ((defns defns))
      (if (null? defns)
        (empty-tenv)
        (extend-tenv-with-module
          (module-definition->name (car defns))
          (expand-module-type-in-tenv
            (module-definition->module-type (car defns))
            tenv)
          (recur (cdr defns)))))))
```

When we process a set of definitions, we must check each expression in the type environment established by the type definitions. For example in the module body from example 13.,

```
[type t = int
 f = proc (x : t) -(x,1)]
```

we must check the definition of `f` in a type environment in which `t` is bound to the type `int`. To accomplish this, we modify `defns-to-decls` to recur with a bigger type environment when it sees a type definition. We make sure to expand the type before we record it in the type environment, so that all types in the type environment are expanded.

```
(define defns-to-decls
  (lambda (defns tenv)
    (if (null? defns)
      '()
      (cases definition (car defns)
        (val-definition (name exp)
          (let ((ty (expand-answer
                      (type-of exp tenv (empty-subst)))))
            (cons
              (val-declaration name ty)
              (defns-to-decls
                (cdr defns)
                (extend-tenv name ty tenv)))))
        (type-definition (name ty)
          (cons
            (type-abbrev-declaration name ty)
            (defns-to-decls
              (cdr defns)
              (extend-tenv-with-type-abbrev name
                (expand-type-in-tenv ty tenv)
                tenv))))))))
```

Next, we need to modify `<:-declarations` to handle the two new kinds of declarations. The first difference we must deal with is that there are now scoping relations inside a set of declarations. For example, if we are comparing

```
[type t = int  x : bool y : t] <: [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. We do this by calling `extend-tenv-with-declaration`, which takes a declaration and translates it to an appropriate extension of the type environment. We always use `decls1` for the extension, because anything in `decls2` must be in `decls1` as well.

```
(define <:-declarations
  (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)
            (if (<:-declaration (car decls1) (car decls2) tenv)
              (<:-declarations (cdr decls1) (cdr decls2)
                (extend-tenv-with-declaration
                  (car decls1) tenv))
              #f)
            (<:-declarations (cdr decls1) decls2
              (extend-tenv-with-declaration
                (car decls1) tenv))))))))
```

```
(define extend-tenv-with-declaration
  (lambda (decl tenv)
    (cases declaration decl
      (type-abbrev-declaration (name ty)
        (extend-tenv-with-type-abbrev name ty tenv))
      (val-declaration (name ty)
        (extend-tenv name ty tenv))
      (abstract-type-declaration (name)
        (extend-tenv-with-abstype name tenv)))))
```

Now we get to the key question: how do we compare declarations?

There are four ways in which a pair of declarations can match:

$$\textsc{<:-VAL-VAL}\quad \frac{ty_1 <: ty_2}{(var : ty_1) <: (var : ty_2)}$$

$$\textsc{<:-ABS-ABS}\quad \frac{}{(\texttt{abstract-type}\ t) \ <: \ (\texttt{abstract-type}\ t)}$$

$$\textsc{<:-ABBREV-ABS}\quad \frac{}{(\texttt{type-abbrev}\ t = ty) \ <: \ (\texttt{abstract-type}\ t)}$$

$$\textsc{<:-ABBREV-ABBREV}\quad \frac{ty_1 \equiv ty_2}{(\texttt{type-abbrev}\ t = ty_1) \ <: \ (\texttt{type-abbrev}\ t = ty_2)}$$

Two value declarations are related if and only if they declare the same variable and the same type. Two abstract-type declarations are related if and only if they declare the same variable. An type abbreviation always matches an abstract-type declaration of the same name. 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,

$$(\texttt{abstract-type}\ t)\ \textbf{NOT}{<:}\ (\texttt{type-abbrev}\ t = ty)$$

because the value with an abstract type may have some type other than `int`, and we are not allowed to know.

Two type abbreviations match if they define the same type name and the two definitions have equal expansions. Finally, a type declaration of any kind cannot match a value declaration.

This gives us the following code.

```
(define <:-declaration
  (lambda (decl1 decl2 tenv)
    (cond
      ((and (val-decl? decl1) (val-decl? decl2))
       (<:-type (decl->type decl1) (decl->type decl2) tenv))
      ((and (abstype-decl? decl1) (abstype-decl? decl2))
       #t)
      ((and (type-abbrev-decl? decl1) (type-abbrev-decl? decl2))
       (equiv-type? (decl->type decl1) (decl->type decl2) tenv))
      ((and (type-abbrev-decl? decl1) (abstype-decl? decl2))
       #t)
      (else #f))))

(define equiv-type?
  (lambda (ty1 ty2 tenv)
    (equal?
      (expand-type-in-tenv ty1 tenv)
      (expand-type-in-tenv ty2 tenv))))
```

In `<:-type` we expand the types before comparing them; otherwise we proceed as before.

```
(define <:-type
  (lambda (ty1 ty2 tenv)
    (let recur ((ty1 (expand-type-in-tenv ty1 tenv))
                (ty2 (expand-type-in-tenv ty2 tenv))
                (subst (empty-subst)))
      (cond
        ...))))
```

Similarly, in the unifier, we add a call to `expand-type-in-tenv`, so that types that come from the program text (for example, from a bound-variable or `letrec` declaration) are appropriately expanded. The main portion of the unifier is unchanged, except that we introduce a local procedure to avoid re-expanding the types at every recursive call.

unifier `: type * type * tenv * subst * exp -> subst`
usage: `finds the smallest extension of subst`
```
   that unifies the expansions of ty1[subst] and ty2[subst].
   Raises an error if there is no such unifier.
```

```
(define unifier
  (lambda (ty1 ty2 tenv subst exp)
    (let recur
      ((ty1 (expand-type-in-tenv (apply-subst subst ty1) tenv))
       (ty2 (expand-type-in-tenv (apply-subst subst ty2) tenv))
       (subst subst))
      ...)))
```

Last, we need to modify `type-of` to handle qualified variable references. This modification was straightforward in section 2, so we did not discuss it there. Now, however, it is more interesting.

Given a qualified variable reference `from` *m* `take` *var*, we first find the type of *m* in the current type environment. If *m* is an ordinary module, then we look to see if *var* is bound by value declaration in the set of declarations exported by *m*. If so, we wrap its named types with *m*, as we did in `expand-qualified-type-in-tenv`. For example, if `m1` is declared by

```
[abstract-type t
 type-abbrev u = (t -> t)
 val f : (t -> u)]
```

then `from m1 take f` should have type

```
(from m1 take t -> (from m1 take t -> from m1 take t))
```

Here's the code.

```
(define type-of
  (lambda (exp tenv subst)
    (cases expression exp
      (qualified-var-ref (m-name comp-name)
        (let ((m-type
                (expand-module-type-in-tenv
                  (lookup-module-name-in-tenv tenv m-name)
                  tenv)))
          (cases module-type m-type
            (env-module-type (decls)
              (an-answer
                (wrap-named-types-with-m-name
                  (lookup-val-decl-in-decls comp-name decls)
                  m-name)
                subst))
              )))
      ...)))
```

7.5 Parameterized Modules and Explicit Dependencies

So far, all of the programs in MODULES have had a fixed set of modules. Sometimes we say the dependencies are *hard-coded*.

In general, such hard-coded dependencies are poor software-engineering practice. Sometimes we may want to experiment with different implementations of the same interface. For example, in our interpreters we might want to be able to use different implementations of environments, or different implementations of the store or of mutable pairs. Or we might want to build a database application and use it with different database backends.

To allow this, we would like to write a module transformer that takes, say, a module that implements the mutable pairs interface, and produces a module that implements the interpreter interface. We call such a module transformer a *parameterized module*. A parameterized module is much like a procedure, except that it works with module values, rather than with expressed values.

We next present a sequence of examples showing parameterized modules in action. We will use implementations of arithmetic, as in section 2, as a running example, but in a richer language, we would use modules for larger bundles of functionality, e.g. environments, hash tables, etc. We start with two implementations of arithmetic.

Example 20. The program

```
module int1
 interface [abstract-type t
           zero : t
           succ : (t -> t)
           pred : (t -> t)
           is-zero : (t -> bool)]
 implementation [type t = int
                zero = 0
                succ = proc(x : ?) -(x,-5)
                pred = proc(x : ?) -(x,5)
                is-zero = proc (x : ?) zero?(x)]

 import int1
 let zero = from int1 take zero
 in let succ = from int1 take succ
 in (succ (succ zero))
```

has type `from int1 take t`. It has value 10, but we can manipulate this value only through the procedures that are exported from `int1`. This module represents the integer k by the expressed value $5 * k$. So $\lceil k \rceil = 5 * k$.

Example 21. In this module, $\lceil k \rceil = -3 * k$.

```
module int2
 interface [abstract-type t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)]
 implementation [type t = int
                 zero = 0
                 succ = proc(x : ?) -(x,3)
                 pred = proc(x : ?) -(x,-3)
                 is-zero = proc (x : ?) zero?(x)]

 import int2
 let z = from int2 take zero
 in let s = from int2 take succ
 in (s (s z))
```

has type `from int2 take t` and has value -6.

Example 22. In the preceeding 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 int1
 interface [abstract-type t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)]
 implementation [type t = int
                 zero = 0
                 succ = proc(x : ?) -(x,-5)
                 pred = proc(x : ?) -(x,5)
                 is-zero = proc (x : ?) zero?(x)
                 ]
import int1
let z = from int1 take zero
in let s = from int1 take succ
in let p = from int1 take pred
in let z? = from int1 take is-zero
in letrec int to-int (x : from int1 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 23. Here is the same technique used with the implementation `int2`: arithmetic,

```
module int2
 interface [abstract-type t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)]
 implementation [type t = int
                 zero = 0
                 succ = proc(x : ?) -(x,3)
                 pred = proc(x : ?) -(x,-3)
                 is-zero = proc (x : ?) zero?(x)
                 ]
 import int2
 let z = from int2 take zero
 in let s = from int2 take succ
 in let p = from int2 take pred
 in let z? = from int2 take is-zero
 in letrec int to-int (x : from int2 take t) =
              if (z? x) then 0
                 else -((to-int (p x)), -1)
 in (to-int (s (s z)))
```

also has type `int` and has value 2.

We've used the same client `to-int` on two different implementations of arithmetic. So let's write `to-int` as our first parameterized module.

Example 24. The declaration

```
module to-int-maker
 interface
  ((m1 : [abstract-type t
          zero : t
          succ : (t -> t)
          pred : (t -> t)
          is-zero : (t -> bool)])
    => [to-int : (from m1 take t -> int)])
 implementation
  module-proc
   (m1 : [abstract-type t
          zero : t
          succ : (t -> t)
          pred : (t -> t)
          is-zero : (t -> bool)])
   [to-int
     = let z? = from m1 take is-zero
       in let p = from m1 take pred
       in letrec int to-int (x : from m1 take t)
         = if (z? x)
            then 0
            else -((to-int (p x)), -1)
       in to-int]
```

defines a parameterized module. The interface says that this module takes as an module `m1` that implements the interface of arithmetic, and produces another module that exports a `to-int` procedure that converts `m1`'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!

Next, let's look at a few examples of `to-int` in action:

Example 25.

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

module int1 ... as before ...

module int1-to-int
 interface [to-int : (from int1 take t -> int)]
 implementation
  import to-int-maker
  import int1
  (to-int-maker int1)

import int1
import int1-to-int
let two1 = (from int1 take succ
            (from int1 take succ
             from int1 take zero))
in (from int1-to-int take to-int
    two1)
```

has type `int` and value 2. Here we first define the modules `to-int-maker`, and `int1`. Then we apply `to-int-maker` to `int1`, getting the module `int1-to-int`, which exports the value `to-int`.

Example 26.

```
module to-int-maker ... as before ...
module int1 ... as before ...
module int2 ... as before ...

module int1-to-int
 interface [to-int : (from int1 take t -> int)]
 implementation
  import int1
  import to-int-maker
  (to-int-maker int1)

module int2-to-int
 interface [to-int : (from int2 take t -> int)]
 implementation
  import int2
  import to-int-maker
  (to-int-maker int2)

import int1
import int1-to-int
import int2
import int2-to-int

let s1 = from int1 take succ
in let z1 = from int1 take zero
in let to-int1 = from int1-to-int take to-int
in let s2 = from int2 take succ
in let z2 = from int2 take zero
in let to-int2 = from int2-to-int take to-int

in let two1 = (s1 (s1 z1))
in let two2 = (s2 (s2 z2))
in -((to-int1 two1), (to-int2 two2))
```

has type `int` and value 0.

If we had replaced `(to-int2 two2)` by `(to-int2 two1)`, the program would not be well-typed, because `to-int2` expects an argument from the `int2` representation of arithmetic, and `two1` is a value from the `int1` representation of arithmetic.

7.5.1 Implementation

7.5.1.1 Syntax

Adding parameterized modules to our language is much like adding procedures. A parameterized module has a module type that is much like a `proc` type.

```
(module-type
  ("(" "(" identifier ":" module-type ")" "=>" module-type ")")
  proc-module-type)
```

These types are different from the types of ordinary procedures, not only because they denote functions from module values to module values, but also because they give a name to the input to the function. This is necessary, because the type of the output may depend on the type of the input, as in the type of `to-int-maker`:

```
((m1 : [abstract-type t
         zero : t
         succ : (t -> t)
         pred : (t -> t)
         is-zero : (t -> bool)])
  => [to-int : (from m1 take t -> int)])
```

`to-int-maker` takes a module `m1` and produces a module whose type depends not just on the type of `m1`, which is fixed, but on its value. Types such as this are called *dependent types*.

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.

```
(module-body
  ("module-proc"
    "(" identifier ":" module-type ")"
    module-body)
  proc-module-body)

(module-body
  (identifier)
  var-module-body)

(module-body
  ("(" identifier identifier ")")
  app-module-body)
```

7.5.1.2 The Interpreter

For the interpreter, we add a new kind of module value, analogous to a procedure.

```
(define-datatype module-value module-value?
  (env-module-value
    (bindings environment?))
  (proc-module-value
    (bvar symbol?)
    (body module-body?)
    (saved-env environment?)))
```

We extend `value-of-module-body` to handle the new possibilities for a module body. This code is much like that for variable references and procedure calls in expressions.

```
(define value-of-module-body
  (lambda (m-body env)
    (cases module-body m-body
      (defns-module-body (defns) ...)
      (var-module-body (var)
        (lookup-module-in-env var env))
      (proc-module-body (m-var m-type m-body)
        (proc-module-value m-var m-body env))
      (app-module-body (rator rand)
        (let ((rator-val (lookup-module-in-env rator env))
              (rand-val (lookup-module-in-env rand env)))
          (cases module-value rator-val
            (proc-module-value (m-var m-body env)
              (value-of-module-body m-body
                (extend-env-with-module m-var rand-val env)))
            (else (eopl:error 'value-of-module-body
                    "can't apply non proc-module-value ~s"
                    rator-val))))))))
```

7.5.1.3 The Checker

We can write down rules like the ones in the last lecture for our new kinds of module bodies. We write $(\triangleright\ \mathit{tenv}\ \mathit{body})$ `=` ty instead of (`type-of-module-body` $\mathit{tenv}\ \mathit{body}$) `=` ty in order to make the rules fit on the page.

$$(\triangleright\ m\ \ \mathit{tenv}) \ \texttt{=}\ \mathit{tenv}(m)$$

$$\frac{(\triangleright\ \mathit{body}\ \ [m\texttt{=}ty_1]\mathit{tenv})\ \texttt{=}\ ty_2}{(\triangleright\ \texttt{(m-proc}\ (m\texttt{:}ty_1)\ \ \mathit{body}\ \ \mathit{tenv})\ \texttt{=}\ ((m\texttt{:}ty_1)\ \texttt{=>}\ ty_2)}$$

A module variable reference 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.

The rule for application of a parameterized module is

$$\frac{tenv(m_1) = \texttt{((}m\texttt{:}ty_2\texttt{')}\ \texttt{=>}\ ty_3\texttt{)} \qquad tenv(m_2) = ty_2 \qquad ty_2 <: ty_2'}{\texttt{(}\triangleright\ \texttt{(}m_1\ \ m_2\texttt{)}\ \ tenv\texttt{)}\ \texttt{=}\ ty_3\texttt{[}m_2\texttt{/}m\texttt{]}}$$

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

First, the type of the operand (ty_2 in the rule below) need not be exactly the same as the parameter type (ty_2'). We require only that $ty_2 <: ty_2'$. This is sufficient, since $ty_2 <: ty_2'$ implies that any module that satisfies the interface ty_2 also satisfies the interface ty_2', and is therefore an acceptable argument to the module procedure.

Second, we substitute the operand m_2 for m in the result type ty_3. This is where we use the dependent types in our type system. For example, consider `to-int-maker`, which has type

```
((m1 : [abstract-type t
        zero : t
        succ : (t -> t)
        pred : (t -> t)
        is-zero : (t -> bool)])
  => [to-int : (from m1 take t -> int)])
```

When we apply `to-int-maker` to `int1`, as we did in example 26., we get a module with type

```
[to-int : (from int1 take t -> int)]
```

When we apply it to `int2`, we get a module with type

```
[to-int : (from int2 take t -> int)]
```

From these rules, it is easy to write down the code for `type-of-module-body`.

```
;; module-body * tenv -> module-type
(define type-of-module-body
  (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)
        (env-module-type
          (defns-to-decls defns tenv)))
      (app-module-body (rator-id rand-id)
        (let ((rator-type
                (lookup-module-name-in-tenv tenv rator-id))
              (rand-type
                (lookup-module-name-in-tenv tenv rand-id)))
          (cases module-type rator-type
            (env-module-type (type-env)
              (eopl:error 'type-of-module-body
                "attempt to apply non-parameterized module ~s"
                rator-id))
            (proc-module-type (param-name expected-type
                                result-type)
              (if (not (<:-module-type
                         rand-type expected-type tenv))
                (raise-bad-module-application-error
                  expected-type rand-type m-body))
              (rename-in-module-type
                result-type param-name rand-id))
            (else (eopl:error 'type-of-module-body
                    "unknown module type ~s"
                    rator-type)))))
      (proc-module-body (m-name m-type m-body)
        (let ((body-type
                (type-of-module-body m-body
                  (extend-tenv-with-module m-name m-type tenv))))
          (proc-module-type m-name m-type body-type))))))
```

This code uses the procedure `rename-in-module-type` to perform the substitution in the result type.

Last, we extend `<:-module-type` to handle the new types. The rule for function module types is

$$\frac{ty_1' <: ty_1 \qquad ty_2[m''/m'] <: ty_2'[m''/m]}{\texttt{((}m\texttt{:}ty_1\texttt{) => }ty_2\texttt{)} <: \texttt{((}m'\texttt{:}ty_1'\texttt{) => }ty_2'\texttt{)}}$$

For $\texttt{((}m\texttt{:}ty_1\texttt{) => }ty_2\texttt{)}$ to be a subtype of $\texttt{((}m'\texttt{:}ty_1\texttt{') => }ty_2\texttt{')}$, it must be the case that any module m_1 of the first type can be used in place of a module m_2 of the second type. That means that any argument to m_2 can be passed to m_1, and any module value that m_1 produces can be used in place of a value of m_2.

For the first requirement, we insist that $ty_1' <: ty_1$ Note the reversal: we say that subtyping is *contravariant* in the parameter type.

What about the result types? We might ask that $ty_2 <: ty_2'$, but that is not quite right. ty_2 may have instances of the module variable m in it, and ty_2' may have instances of m' in it. So to compare them, we rename both m and m' to some fresh module variable m''. Once we do that, we can compare them sensibly. This leads to the requirement $ty_2[m''/m'] <: ty_2[m''/m]$.

The code to decide this relation is relatively straightforward.

We call `expand-module-type-in-tenv` to make sure all of the types are expanded, and when deciding $ty_2[m''/m'] <: ty_2[m''/m]$ we extend the type environment with the appropriate type for m''.

```
(define <:-module-type
  (lambda (m-type1 m-type2 tenv)
    (let ((m-type1
            (expand-module-type-in-tenv m-type1 tenv))
          (m-type2
            (expand-module-type-in-tenv m-type2 tenv)))
    (cases module-type m-type1
      (env-module-type (decls1)
        (cases module-type m-type2
          (env-module-type (decls2)
            (<:-declarations decls1 decls2 tenv))
          (else #f)))
      (proc-module-type (param-name1 param-type1 m-type1)
        (cases module-type m-type2
          (proc-module-type (param-name2 param-type2 m-type2)
            (let ((new-name (fresh-module-name param-name1)))
              (let ((m-type1
                      (rename-in-module-type
                        m-type1 param-name1 new-name))
                    (m-type2
                      (rename-in-module-type
                        m-type2 param-name2 new-name)))
                (and
                  (<:-module-type param-type2 param-type1 tenv)
                  (<:-module-type m-type1 m-type2
                    (extend-tenv-with-module
                      new-name param-type2 tenv))))))
          (else #f)))))))
```

And now we're done. Go have a hot fudge sundae, with ice cream, fudge, and nuts, all from different containers. Don't worry about how any of the pieces are constructed, so long as they taste good!