# Chapter 8. Modules
# (Essentials of Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

# Outline

This chapter discusses modules that we will need when we are to build larger systems with thousands of lines of code.

- ♢ The Simple Module System
- ♢ Modules That Declare Types
- ♢ Module Procedures

# Introduction

When we are to build large systems, it is desirable to have:

- A way to separate the system into self-contained parts, and to document the dependencies between those parts
- A way to control the scope and binding of names
- A way to enforce abstraction boundaries
- A way to combine these parts flexibly so that a single part may be reused in different contexts.

$\Rightarrow$ Modules & type systems for modules (to create and enforce abstraction boundaries)

# Introduction (Cont.)

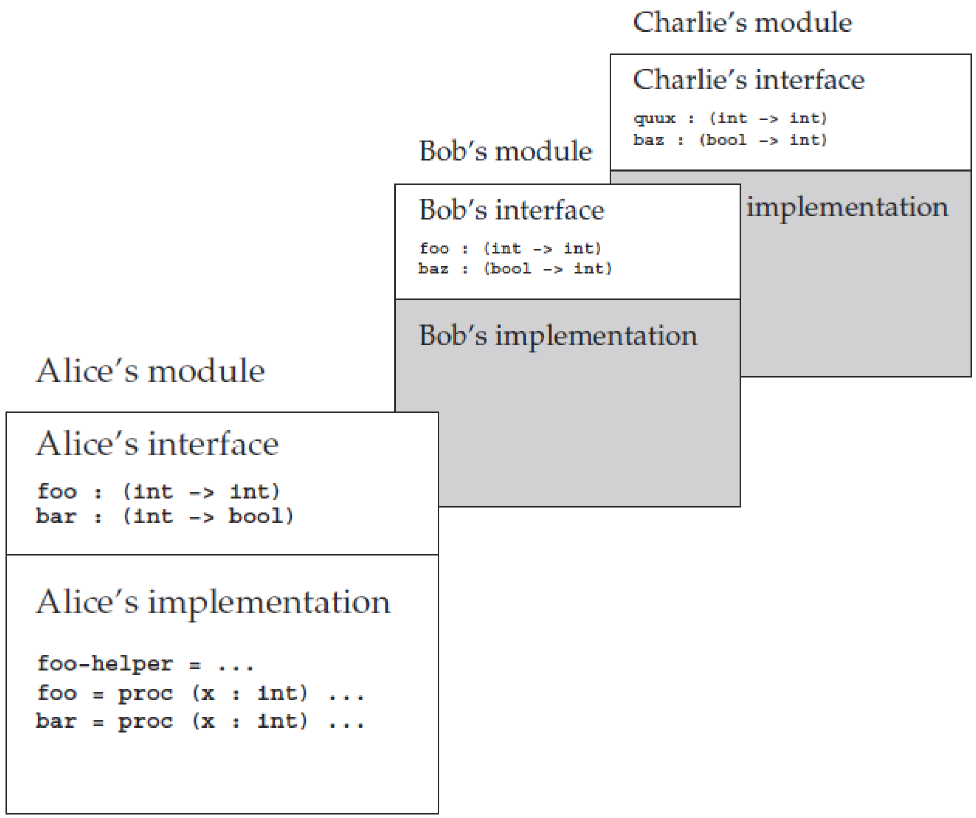
A program = a sequence of module definitions + a main expression

- Each definition binds a name to a module
- A created module is either
  - a simple module (a set of bindings like an environment) or
  - a module procedure (taking a module and producing another)
- Each module has an interface as:
  - a simple interface (listing the module bindings and their types)
  - a module procedure interface (specifying
    - the interface of the argument module interface and
    - the interface of the result module)

Module interfaces determine the ways in which modules can be combined.

# 8.1 The Simple Module System

Alice's view of the three modules in the project



# 8.1.1 Examples

SIMPLE-MODULES: a language with only simple modules (and no module procedures)

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

# 8.1.1 Examples

Some of the scopes for a simple module

```
module m1
 interface
  [ a : int
    b : int
    c : int ]
 body
  [ a = 33
    x = -(a,1)
    b = -(a,x)
    c = -(x,b) ]
```

*scope of* `a = 33`

*scope of*

```
from m1 take a
from m1 take b
from m1 take c
```

```
let a = 10
in -(from m1 take a,
     a)
```

cf. let* scoping (Exercise 3.17)

# 8.1.1 Examples (Cont.)

In the previous example,

- The body *implements* the interface.
  - The interface *offers* (or *advertises* or *promises*) three integer values.
  - The body *supplies* (or *provoides* 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.

## 8.1.1 Examples (Cont.)

In the previous example (Cont.),

- The scoping technology does not scale to the module program.
  - *Qualified variables* vs. simple variables
    - 'from m1 take x' vs. x
- 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
  - Everything else is *outside* the abstraction boundary.
    - 'from m1 take x' is not in scope

# 8.1.1 Examples (Cont.)

An ill-typed module

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

On ordering of bindings

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

Two modules in a correct order

- ```
  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)
  ```
  ```

# 8.1.1 Examples (Cont.)

The two modules in an incorrect order

- 

```
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)
```

# 8.1.2 Implementing the Simple Module System

Syntax - module and interface

- Program ::= $\{$ModuleDefn$\}^*$ Expression
  $\boxed{\text{a-program (m-defs body)}}$

- ModuleDefn ::= module Identifier interface Iface body Module-Body
  $\boxed{\text{a-module-definition (m-name expected-iface m-body)}}$
- Iface ::= [ $\{$Decl$\}^*$ ]
  $\boxed{\text{simple-iface (decls)}}$
- Decl ::= Identifier : Type
  $\boxed{\text{val-decl (var-name ty)}}$

# 8.1.2 Implementing the Simple Module System (Cont.)

Syntax - module body

- ModuleBody ::= [ $\{$Defn$\}^*$ ]
  $\boxed{\text{val-defn (var-name exp)}}$
- Defn ::= Identifier = Expression
  $\boxed{\text{val-defn (var-name exp)}}$
- Expression ::= from Identifier take Identifier
  $\boxed{\text{qualified-var-exp (m-name var-name)}}$

# 8.1.2 Implementing the Simple Module System (Cont.)

The Interpreter

- Evaluation of a module body will produce a *module*, which is an environment consisting of all the bindings exported by the module.

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

(define-datatype environment environment?
  .... as before ...
  (extended-env-with-module
    (m-name symbol?)
    (m-val typed-module?)
```

```
(saved-env environment?)))
```

# 8.1.2 Implementing the Simple Module System (Cont.)

```
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))
```

```
(extended-env z (num-val 99)
 (exnteded-env-with-module
  m2
  (extended-env a (num-val 66)
   (extended-env b (num-val 77)
    (empty-env)))
  (empty-env))
  (extended-env-with-module
   m1
   (extended-env a (num-val 33)
    (extended-env b (num-val 44)
     (extended-env c (num-val 55))))
   (empty-env))
  (empty-env))
```

# 8.1.2 Implementing the Simple Module System (Cont.)

Procedures for the interpreter in Figure 8.3 and 8.4

- lookup-qualified-var-in-env : looking up the module in the current environment and then the variable in the module environment
- value-of-program : adding module definitions to the environment
  - add-module-defns-to-env
- value-of-module-body : producing an environment containing only the bindings produced by the definitions
  - defns-to-env

# 8.1.2 Implementing the Simple Module System (Cont.)

The Checker

- Making sure that each module body satisfies its interface, and that each variable is used consistently with its type.
- Our language follows let* scoping putting into scope qualified variables for each of the bindings exported by the module.

## 8.1.2 Implementing the Simple Module System (Cont.)

In type environment, each module name is bound to its interface.

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

## 8.1.2 Implementing the Simple Module System (Cont.)

Procedures for the interpreter in Figure 8.5, 8.6, and 8.7.

- looku-qualified-var-in-tenv : looking up the type of the module name and then the type of the variable
- type-of-program : adding module interfaces to the type environment
  - add-module-defns-to-tenv
- <:-iface : checking if the interface produced by the module body matches the advertised interface
  - <:-decls

# 8.1.2 Implementing the Simple Module System (Cont.)

An interface produced by the module body

$$\begin{bmatrix} a = 33 \\ x = -(a,1) \\ b = -(a,x) \\ c = -(x,b) \end{bmatrix} \qquad \begin{bmatrix} a : \mathrm{int} \\ x : \mathrm{int} \\ b : \mathrm{int} \\ c : \mathrm{int} \end{bmatrix}$$

The produced interface matches the advertise interface

$$\begin{bmatrix} a : \mathrm{int} \\ x : \mathrm{int} \\ b : \mathrm{int} \\ c : \mathrm{int} \end{bmatrix} <: \begin{bmatrix} a : \mathrm{int} \\ b : \mathrm{int} \\ c : \mathrm{int} \end{bmatrix}$$

## 8.1.2 Implementing the Simple Module System (Cont.)

The procedure <:-decls in Figure 8.7

- If decls1 and decls2 are two sets of declarations,
  - decls1 <:- decls2 if and only if any module that supplies bindings for the declarations in decls1 also supplies bindings for the declarations in decls2.

# 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.

## 8.2.1 Examples

Alice provides interfaces for pairs of integers, representing the x- and y- coordinates of a point.

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

Using a module of this interface,

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

```
        proc (p2 : point) ...
    ... ]
```

# 8.2.1 Examples (Cont.)

Bob can write a procedure depending on the implementation of pairs while Alice decides to change the representation of points so that the the y-coordinate is in the first component.

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

Alice can solve her problem by making point an *opaque* data type.

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

```
get-x : (point -> int)
```

Then Bob can no longer manipulate points using any procedures other than the ones in Alice's interface.

# 8.2.1 Examples: Transparent Types

Transparent type declarations (also called concrete type declarations or *type abbreviations*)

```
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))
```

# 8.2.1 Examples: Opaque Types

Opaque type declarations (sometimes called *abstract* types)

```
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))
```

-(x,0) is ill-typed!

# 8.2.1 Examples: Opaque Types (Cont.)

This is the abstraction boundary. The definition of the opaque type t is hidden from the rest of the program.

```
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)
```

# Example 8.8 8.13

# 8.2.2 Implementation: Syntax

Transparent and opaque type decls and qualified type references

Syntax

- Type ::= Identifier
  $\boxed{\text{named-type (name)}}$
- Type ::= from Identifier take Identifier
  $\boxed{\text{qualified-type (m-name t-name)}}$
- Decl ::= opaque Identifier
  $\boxed{\text{opaque-type-decl (t-name)}}$
- Decl ::= transparent Identifier = Type
  $\boxed{\text{transparent-type-decl (t-name ty)}}$
- Defn ::= type Identifier = Type
  $\boxed{\text{type-defn (name ty)}}$

# 8.2.2 Implementation: Interpreter

Interpreter

- 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

# 8.2.2 Implementation: The Checker

The Checker

To handle opaque and transparent types in a systematic way, we use expanded type as

- Type ::= int | bool | from m take t | Type $\rightarrow$ Type

Our type environments will bind each named type or qualified type to an expanded type.

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

```
(saved-tenv type-environment?)))
```

# 8.2.2 Implementation: The Checker (Cont.)

A procedure, expand-type (ty, tenv) = expanded-type

- In type-of in the checker, we always use (extend-tenv var (expanded-type ty tenv) tenv)
- When we process a list of definitions with defns-to-decls, we expand its right-hand side and add it to the type environment.
- Where we add a module to the type environment, in add-module-defns-to-tenv, we need to expand the interface to the type environment (Figure 8.9)
  - expand-iface (Figure 8.9)
  - This calls expand-decls (Figure 8.10)

## 8.2.2 Implementation: The Checker (Cont.)

Lastly, we modify <:-decls to handle the two new kinds of declarations as (Figure 8.11):

- They are both value declarations, and their types match.
- They are both opaque type declarations.
- They are both transparent type declarations, and their defs match.
- decl1 is a transparent type declaration, and decl2 is an opaque type declaration.
  - (transparent t = int) $<:$ (opaque t)
  - (opaque t) $\not<:$ (transparent t = int)

equiv-type? compares two types in their expanded form. (Figure 8.12)

# 8.3 Module Procedures

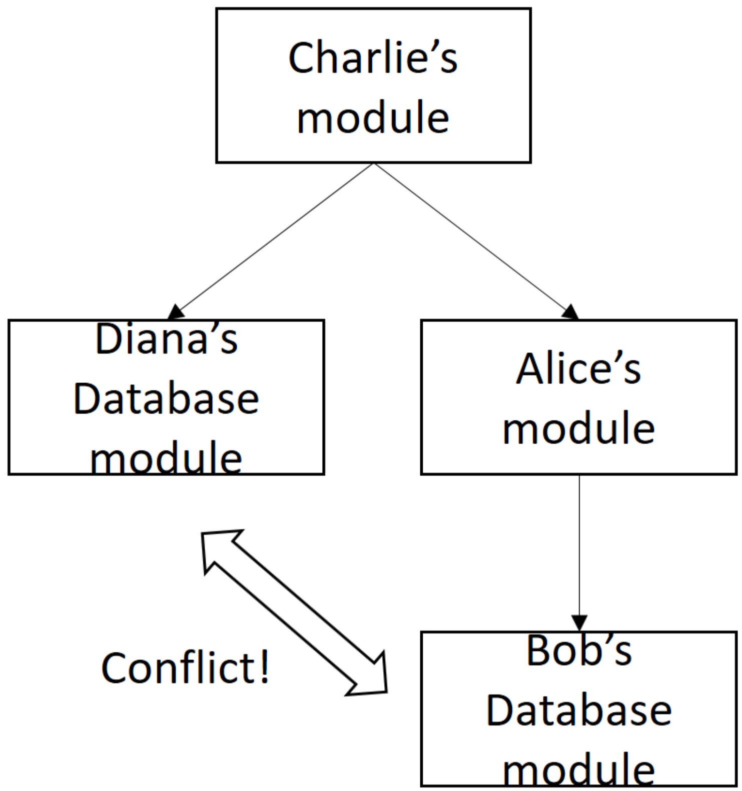
In OPAQUE-TYPES, the programs have a fixed set of dependencies, which are sometimes hard-coded.

Hard-coded dependencies lead to bad program design because they make it difficult to reuse modules.

A solution is module procedures.

# 8.3.1 Examples

Charlie wants to use Alice's module with Diana's database module.
How could you rewrite Alice's module in a modular way?



## 8.3.1 Examples (Cont.)

To make this possible in a modular way, Alice rewrites her code using *module procedures* (sometimes called parameterized modules).

A module procedure is like a procedure, except that it works with modules, rather than with expressed values.

- modules vs. expressed values
- interfaces (of modules) vs. types (of expressed values)

Q. Explain how a module procedure solves the conflict in the previous example.

PROC-MODULES, a language with module procedures

## 8.3.1 Examples (Cont.)

A module procedure, Alices-point-builder begins with

```
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) ] )
 body ...
```

- In Alices-points, (Alices-point-builder Bobs-db-module)
- In Charlies-points, (Alices-point-builder Dianas-db-module)

# 8.3.1 Examples (Cont.)

```
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) ])
 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 ... ]
```

cf. database vs. m

# 8.3.1 Examples (Cont.)

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)
```

# 8.3.2 Implementation

The syntax

- Iface ::= ((Identifier : Iface) => Iface)
  $\boxed{\text{proc-iface (param-name param-iface result-iface)}}$

Two differences from normal function types

- A module interface describes functions from module values to module values.
- This module interface gives a name to the input to the function because the interface of the output may depend on the values of the input. (cf. *dependent types*)
  - (See the next slide)

# 8.3.2 Implementation (Cont.)

A module interface gives a name to the input to the function because the interface of the output may depend on the values of the input. (cf. *dependent types*)

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

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

# 8.3.2 Implementation (Cont.)

The syntax for module body

- ModuleBody ::= [ $\{$Defn$\}^*$ ] $\boxed{\text{val-defn (var-name exp)}}$

is extended with

- ModuleBody ::= module-proc (Identifier : Iface) ModuleBody
  $\boxed{\text{proc-module-body (m-name m-type m-body)}}$
- ModuleBody ::= Identifier
  $\boxed{\text{var-module-body (m-name)}}$
- ModuleBody ::= (Identifier Identifier)
  $\boxed{\text{app-module-body (rator rand)}}$

cf. LC-exp (lambda calculus exprs.: lambda-exp, var-exp, app-exp)

# 8.3.2 Implementation (Cont.)

Module values

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

In the interpreter, *value-of-module-body* takes a module body with an environment and produces a module value.

- Figure 8.13

# 8.3.2 Implementation (Cont.)

The checker: the rules for typing new module bodies (Fig. 8.15)

(IFACE-M-VAR) interface-of m tenv = tenv(m)

(IFACE-M-PROC)

$$\frac{\text{interface-of body [m=i1]tenv = i2}}{\text{interface-of (m-proc (m:i1) body) tenv = (m:i1) =>i2}}$$

(IFACE-M-APP)

$$\frac{\text{tenv(m1) = (m:i1) =>i1'} \qquad \text{tenv(m2) = i2} \qquad \text{i2 <: i1}}{\text{interface-of (m1 m2) tenv = (m:i1) =>i2}}$$

Q. What is the definition of '<:' for procedure interfaces (m:i)=>i'?

- The notation $\triangleright\ body\ tenv$ in the textbook, instead of interface-of body tenv

# 8.3.2 Implementation (Cont.)

An extension of i1 $<:$ i2 with interfaces for module procedures (Fig. 8.16)

$$\frac{\text{i2} <\text{i1} \qquad \text{i1'[m'/m1]} <\text{i2'[m'/m2]} \qquad \text{m' not in i1' or i2'}}{\text{(m1:i1)} => \text{i1'} \quad <: \quad \text{(m2:i2)} => \text{i2'}}$$

Note that the subtyping is *contravariant* in the parameter type.

- (m1:[x:int]) $=>$[z:int] $<:$ (m1:[x:int,y:int]) $=>$[z:int]
- (m1:[x:int]) $=>$[y:int,z:int] $<:$ (m1:[x:int]) $=>$[z:int]
- (m1:[x:int]) $=>$[w:int,z:int] $<:$ (m1:[x:int,y:int]) $=>$[z:int]