Typechecking Simple Modules

CS496

Modules

For large systems one needs:

1. A good way to separate the system into relatively self-contained parts, and to document the dependencies between those parts.
2. A better way to control the scope and binding of names.
   - Lexical scoping insufficient when programs may be large or split up over multiple sources
3. A way to enforce abstraction boundaries (interface/implem.)
4. A way to combine these parts flexibly, so that a single part may be reused in different contexts.

SIMPLE-MODULES

- Adds a simple module system to REC
- A program consists of a sequence of module definitions followed by an expression to be evaluated.

```
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)
```

SIMPLE-MODULES

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.

```
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)
```

SIMPLE-MODULES

- Each module definition binds a name to a module
- A created module is a set of bindings, much like an environment
- Next we'll see more examples of modules

```
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)
```

Example 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.
- `from m1 take a` and `from m1 take b` are called qualified variables

Example 2

```
module m1
  interface
    [u : bool]
  body
    [u = 33]
4
```

- Type error!
- 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 3

```
module m1
  interface
    [u : int
     v : int]
  body
    [u = 33]
4
```

- Type error!
- The module body must supply bindings for all the declarations in the interface.

Example 4

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

- Type error!
- To keep the implementation simple, our language requires that the module body produce the values in the same order as the interface.

Example 5

```
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`
- Modules have `let*` scoping

Example 6

```
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)
```

- Type error!
- `from m1 take u` is not in scope where it is used in the body of `m2`.

SIMPLE-MODULES: Concrete Syntax

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \{\langle \textit{ModuleDefn} \rangle\}^* \ \langle \textit{Expression} \rangle \\
\\
\langle \textit{ModuleDefn} \rangle & ::= & \text{module } \langle \textit{Identifier} \rangle \\
 & & \text{interface } \langle \textit{Iface} \rangle \\
 & & \text{body } \langle \textit{ModuleBody} \rangle
\end{array}
$$

SIMPLE-MODULES: Concrete Syntax

$\langle \mathit{ModuleDefn} \rangle ::= \text{module } \langle \mathit{Identifier} \rangle \text{ interface } \langle \mathit{Iface} \rangle \text{ body } \langle \mathit{ModuleBody} \rangle$

- Syntax for the interface of a module:

$$
\begin{array}{lcl}
\langle \mathit{Iface} \rangle & ::= & [\ \{\langle \mathit{Decl} \rangle\}^*\ ] \\
\langle \mathit{Decl} \rangle & ::= & \langle \mathit{Identifier} \rangle : \langle \mathit{Type} \rangle
\end{array}
$$

- Syntax for the body of a module

$$
\begin{array}{lcl}
\langle \mathit{ModuleBody} \rangle & ::= & [\ \{\langle \mathit{Defn} \rangle\}^*\ ] \\
\langle \mathit{Defn} \rangle & ::= & \langle \mathit{Identifier} \rangle = \langle \mathit{Expression} \rangle
\end{array}
$$

SIMPLE-MODULES: Concrete Syntax

$$\begin{array}{lcl} \langle \mathit{Expression} \rangle & ::= & \text{...as before...} \\ \langle \mathit{Expression} \rangle & ::= & \texttt{from}\ \langle \mathit{Identifier} \rangle\ \texttt{take}\ \langle \mathit{Identifier} \rangle \end{array}$$

SIMPLE-MODULES: Abstract Syntax

```
(define-datatype program program?
  (a-program
    (m-defs (listof a-module-definition))
    (body expression?)))

(define-datatype expression expression?
 ;;  ... continues in next slide ...
```

$$\langle \mathit{Program} \rangle \quad ::= \quad \{\langle \mathit{ModuleDefn} \rangle\}^{*} \ \langle \mathit{Expression} \rangle$$

SIMPLE-MODULES: Abstract Syntax

```
(define-datatype a-module-definition a-module-definition?
  (a-module-definition
     (m-name identifier?)
     (expected-iface interface?)
     (m-body module-body?)))
```

$$\langle \mathit{ModuleDefn} \rangle ::= \textsf{module}\ \langle \mathit{Identifier} \rangle\ \textsf{interface}\ \langle \mathit{Iface} \rangle\ \textsf{body}\ \langle \mathit{ModuleBody} \rangle$$

SIMPLE-MODULES: Abstract Syntax

```
(define-datatype interface interface?
  (simple-iface
     (decls (listof declaration?))))

(define-datatype declaration declaration?
  (val-decl
     (var-name identifier?)
     (ty type?)))

```

$$
\begin{array}{lcl}
\langle \mathit{Iface} \rangle & ::= & [\ \{\langle \mathit{Decl} \rangle\}^{*}\ ] \\
\langle \mathit{Decl} \rangle & ::= & \langle \mathit{Identifier} \rangle : \langle \mathit{Type} \rangle
\end{array}
$$

SIMPLE-MODULES: Abstract Syntax

```
(define-datatype module-body module-body?
  (defns-module-body
     (defns (listof definition?))))

(define-datatype definition definition?
  (val-defn
     (var-name identifier?)
     (exp expression?)))
```

$$\begin{array}{lcl} \langle \mathit{ModuleBody} \rangle & ::= & [\ \{\langle \mathit{Defn} \rangle\}^*\ ] \\ \langle \mathit{Defn} \rangle & ::= & \langle \mathit{Identifier} \rangle = \langle \mathit{Expression} \rangle \end{array}$$

Concrete vs Abstract Syntax

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

```
(a-program
 (list
  (a-module-definition
   'm1
   (simple-iface (list (val-decl 'u (int-type)) (val-decl 'v (int-type))))
   (defns-module-body (list (val-defn 'v (const-exp 33)) (val-defn 'u (const-exp 44))))))
 (qualified-var-exp 'm1 'u))
```

Simple Modules

The Interpreter

The Type-Checker

Program Evaluation

- Two part process
  1. Module body evaluation
  2. Body evaluation
- Module body evaluation
  - Will produce an environment consisting of all the bindings exported by the module.
- Body Evaluation
  - Will produce an expressed value.

```
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)
```

Program Evaluation

- Two part process
  1. Module body evaluation
  2. Body evaluation
- Module body evaluation
  - Will produce an environment consisting of all the bindings exported by the module.
- Body Evaluation
  - Will produce an expressed value.

```
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)
```

Program Evaluation

- Two part process
  1. Module body evaluation
  2. Body evaluation
- Module body evaluation
  - Will produce an environment consisting of all the bindings exported by the module.
- Body Evaluation
  - Will produce an expressed value.

```
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)
```

Module Body Evaluation

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

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

Module Body Evaluation

```
(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?)))
```

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

Example

```
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))
```

Module Body Evaluation

Environment extant at line 18

```
#(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))))
```

Module Body Evaluation

Environment extant at line 18

```
#(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))))
```

Module Body Evaluation

Environment extant at line 18

```
#(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))))
```

Body Evaluation

- Same as before except we now have to deal with qualified variables

```
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))))))
```

Program Evaluation

```
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)))))))
```

Program Evaluation

```
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)))))))
```

- Prepares the environment with the result of evaluating all the modules

Program Evaluation

```
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)))))))
```

- `value-of-module-body` evaluates the body of its argument module

Program Evaluation

```
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)))))))
```

- `value-of-module-body` evaluates the body of its argument module

Program Evaluation

```
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))))))))
```

- `value-of-module-body` evaluates the body of its argument module

Program Evaluation

```
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)))))))
```

- value-of-module-body evaluates the body of its argument module

Program Evaluation

```
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)))))))
```

- `value-of-module-body` evaluates the body of its argument module

Program Evaluation

```
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)))))))
```

- `value-of-module-body` evaluates the body of its argument module

Program Evaluation

```
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))))))
```

Evaluating the Body of a Module

```
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)))))))))
```

Simple Modules

The Interpreter

The Type-Checker

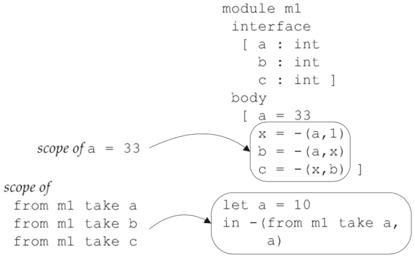
The Type-Checker

Make sure that

- each module body satisfies its interface,
- each variable is used consistently with its type.

The scoping rules:

- Modules follow `let*` scoping, putting into scope qualified variables for each of the bindings exported by the module.
- Declarations and definitions both follow `let*` scoping as well



Type-Checking

1. Obtain type of interface of module body
2. Obtain interface of module
3. Compare them

Let us revisit some examples from above

Example 2

```
module m1
  interface
    [u : bool]
  body
    [u = 33]
4
```

- Type error!
- 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 3

```
module m1
  interface
    [u : int
     v : int]
  body
    [u = 33]
4
```

- Type error!
- The module body must supply bindings for all the declarations in the interface.

Example 7

```
module m1
  interface
    [u : int]
  body
    [v = 2
     u = -(33,v)]
4
```

- Ok!
- `v` is private to module body

Example 43

```
module m1
  interface
    [u : int]
  body
    [v = 2
     u = -(33,v)]
from m1 take v
```

- Type error!
- `v` is private to module body

Example 4

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

- Type error!
- To keep the implementation simple, our language requires that the module body produce the values in the same order as the interface.

Example 6

```
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)
```

- Type error!
- `from m1 take u` is not in scope where it is used in the body of `m2`.

Typing System for SIMPLE-MODULES

- We are going to introduce typing judgements
- Then typing rules
- Finally, we are going to implement a type-checker by using the typing rules as specification

Summary of Typing Judgements

- Judgements for typing expressions

$$\texttt{m\_tenv;tenv} \vdash \texttt{e} :: \texttt{t}$$

- Judgements for typing programs

$$\vdash_{\texttt{M}} \texttt{e} :: \texttt{t}$$

- Judgements for typing list of module declarations

$$\vdash \texttt{M} :: \texttt{m\_tenv}$$

Typing Judgements for Expressions

- Before

$$\texttt{tenv} \vdash \texttt{e} :: \texttt{t}$$

- Now

$$\texttt{m\_tenv;tenv} \vdash \texttt{e} :: \texttt{t}$$

- `tenv` is the standard type environment from before
- `m_tenv` is a module type environment and is required for typing the expression `from m take x`

Typing Programs in Expressions

- Module Type

$$m[u_1 : t_1, \ldots, u_n : t_n]$$

- Module type environment (`m_tenv`)

$$m_1[u_{1,1} : t_1, \ldots, u_{1,n_1} : t_{n_1}] \ldots m_k[u_{k,1} : t_1, \ldots, u_{k,n_k} : t_{n_k}]$$

- If $m[u_1 : t_1, \ldots, u_n : t_n] \in \texttt{m\_tenv}$, then
  - We say $m \in \mathit{Dom}(\texttt{m\_tenv})$
  - $\texttt{m\_tenv}(m, u_i) = t_i$

Typing System for Expressions

$$\frac{m \in \mathit{Dom}(\texttt{m\_tenv}) \quad \texttt{m\_tenv}(m, x) = \texttt{t}}{\texttt{m\_tenv;tenv} \vdash \texttt{from m take x} :: \texttt{t}} \; \mathit{TFromTake}$$

Typing Judgements for Programs

$$\vdash_{\texttt{M}} \texttt{e} :: \texttt{t}$$

- `M` is the list of declared modules

$$\frac{\vdash \texttt{M} :: \texttt{m\_tenv} \qquad \texttt{m\_tenv;empty-tenv} \vdash \texttt{e} :: \texttt{t}}{\vdash_{\texttt{M}} \texttt{e} :: \texttt{t}} \; \textit{TProgram}$$

Typing Judgements for Module Declarations

$$\texttt{m\_tenv}_1 \vdash \texttt{M} :: \texttt{m\_tenv}_2$$

- $\texttt{m\_tenv}_2$ is the type of the list of modules $\texttt{M}$
- $\texttt{m\_tenv}_1$ is the type of the list of modules that $\texttt{M}$ can use
- $list1 \lhd list2$ means that $list1$ is a sublist of $list2$
- $[x_i]_{i \in I} \lhd [y_j]_{j \in J}$ determines an injective, order preserving function $f : I \rightarrow J$

$$\frac{\begin{array}{c} [x_i]_{i \in I} \lhd [y_j]_{j \in J} \\ (\texttt{m\_tenv};[y_1\texttt{:=}s_1]\ldots[y_{j-1}\texttt{:=}s_{j-1}]\,\texttt{tenv} \vdash e_j :: s_j)_{j \in J} \\ (t_i = s_{f(i)})_{i \in I} \\ m[x_i : t_i]_{i \in I}\ \texttt{m\_tenv} \vdash \texttt{M} :: \texttt{m\_tenv} \end{array}}{\texttt{m\_tenv} \vdash \underbrace{m[x_i : t_i]_{i \in I}[y_j = e_j]_{j \in J}}_{\textit{a module declaration}} \quad \underbrace{\texttt{M}}_{\textit{the others}} \quad :: m[x_i : t_i]_{i \in I}\ \texttt{m\_tenv}} \; \textit{TMod}$$

Summary of Typing Judgements

- Judgements for typing expressions

$$\texttt{m\_tenv;tenv} \vdash \texttt{e} :: \texttt{t}$$

- Judgements for typing programs

$$\vdash_{\texttt{M}} \texttt{e} :: \texttt{t}$$

- Judgements for typing list of module declarations

$$\vdash \texttt{M} :: \texttt{m\_tenv}$$

Implementing the Type-Checker

- We will use the typing rules as a guideline
- Rather than having to deal with two different type environments (`m_tenv` and `env` below)

$$\frac{m \in \mathit{Dom}(\texttt{m\_tenv}) \quad \texttt{m\_tenv}(m, x) = t}{\texttt{m\_tenv;tenv} \vdash \texttt{from m take x} :: \texttt{t}}\ \mathit{TFromTake}$$

  we will have just one environment where we can lookup variables and modules

Type Environments

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

Recall from the interpreter:

```
(define-datatype interface interface?
  (simple-iface
     (decls (listof declaration?))))

(define-datatype declaration declaration?
  (val-decl
     (var-name identifier?)
     (ty type?)))
```

Type-Checking Programs

```
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))
)))))
```

$$\frac{\vdash_{\mathtt{M}} :: \mathtt{m\_tenv} \quad \mathtt{m\_tenv;empty\text{-}tenv} \vdash \mathtt{body} :: \mathtt{t}}{\vdash_{\mathtt{M}}\ \mathtt{body} :: \mathtt{t}}\ \textit{TProgram}$$

Type-Checking Qualified Variables

`from m take var`

$$\frac{m \in Dom(\texttt{m\_tenv}) \quad \texttt{m\_tenv}(m, var) = t}{\texttt{m\_tenv;env} \vdash \texttt{from m take var} :: \texttt{t}}\ \textit{TFromTake}$$

```
(define type-of
  (lambda (exp tenv)
    (cases expression exp
      (const-exp (num) (int-type))
      ...
      (qualified-var-exp (m-name var-name)
        (lookup-qualified-var-in-tenv m-name var-name tenv))
     ...
```

Type-Checking Qualified Variables

```
from m take var
```

- first lookup `m` in the type environment,
- then lookup 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)
)))))
```

$$\frac{m \in Dom(\texttt{m\_tenv}) \quad \texttt{m\_tenv}(m, var) = t}{\texttt{m\_tenv;env} \vdash \texttt{from m take var} :: \texttt{t}} \; TFromTake$$

Type-Checking Modules

```
module m1
  interface
    [u : int
     z: int]
  body
    [u = 2
     v = -(33,u)
     z = 7]
4
```

Must compare actual and expected types of each module:

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

`<:-decls` is the operation in charge of this

$$\dfrac{\begin{array}{c} [x_i]_{i\in I} \lhd [y_j]_{j\in J} \\ (\texttt{m\_tenv};[y_1 := s_1]\ldots[y_{j-1} := s_{j-1}]\,\texttt{tenv} \vdash e_j :: s_j)_{j\in J} \\ (t_i = s_{f(i)})_{i\in I} \\ m[x_i : t_i]_{i\in I}\ \texttt{m\_tenv} \vdash \texttt{M} :: \texttt{m\_tenv} \end{array}}{\texttt{m\_tenv} \vdash \underbrace{m[x_i : t_i]_{i\in I}[y_j = e_j]_{j\in J}}_{\textit{a module declaration}}\ \underbrace{\texttt{M}}_{\textit{the others}}\ :: m[x_i : t_i]_{i\in I}\ \texttt{m\_tenv}}\ \textit{TMod}$$

Type-Checking Modules

$$\frac{\begin{array}{c}[x_i]_{i\in I} \lhd [y_j]_{j\in J} \\ (\mathtt{m\_tenv}\,;\,[y_1\mathtt{:=}s_1]\ldots[y_{j-1}\mathtt{:=}s_{j-1}]\,\mathtt{tenv} \vdash e_j :: s_j)_{j\in J} \\ (t_i = s_{f(i)})_{i\in I} \\ m[x_i : t_i]_{i\in I}\ \ \mathtt{m\_tenv} \vdash \mathtt{M} :: \mathtt{m\_tenv}\end{array}}{\mathtt{m\_tenv} \vdash \underbrace{m[x_i : t_i]_{i\in I}[y_j = e_j]_{j\in J}}_{\textit{a module declaration}} \quad \underbrace{\mathtt{M}}_{\textit{the others}} \quad :: m[x_i : t_i]_{i\in I}\ \ \mathtt{m\_tenv}}\ \mathit{TMod}$$

$$\frac{}{\mathtt{m\_tenv} \vdash \epsilon :: \mathtt{[]}}\ \mathit{TModE}$$

Type-Checking Programs

```
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))
)))))
```

$$\frac{\vdash_M :: \texttt{m\_tenv} \quad \texttt{m\_tenv;empty-tenv} \vdash e :: \texttt{t}}{\vdash_M \; e :: \texttt{t}} \; \textit{TProgram}$$

```
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))))))))
```

```
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))))))))
```

```
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))))))))
```

```
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))))))))
```

```
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))))))))
```

```
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))))))))))
```

Subtyping Interfaces

```
<:-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))))))))
```

Work delegated to helper function `<:decls`

```
<:-decls::{Listof(Decl),Listof(Decl),Tenv} -> Bool
(define <:-decls
 (lambda (actual expectd tenv)
  (cond
     ((null? expectd) #t)
     ((null? actual) #f)
     (else
       (let ((name1 (decl->name (car actual)))
             (name2 (decl->name (car expectd))))
         (if (eqv? name1 name2)
             (and (equal?
                         (decl->type (car actual))
                         (decl->type (car expectd)))
                  (<:-decls (cdr actual) (cdr expectd)
    tenv))
            (<:-decls (cdr actual) expectd tenv)))))))
```

The Interpreter for SIMPLE-MODULES

- Code available from `http://www.eopl3.com`
- Directory `chapter8/simplemodules`
- Open `top.scm` in Racket
- There are a number of tests in `test-suite.scm`
- You can run them with `run-one`. Eg.

```
> (run-one 'modules-take-one-value)
(num-val 3)
```

Beyond SIMPLE-MODULES

- Opaque types: modules that declare types (not just values)
- Module procedures: modules that take a module as argument and produce another module as result