# 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

$$\begin{array}{lll}
\langle \textit{Program} \rangle & ::= & \{\langle \textit{ModuleDefn} \rangle\}^* \ \langle \textit{Expression} \rangle \\[1em]
\langle \textit{ModuleDefn} \rangle & ::= & \texttt{module} \ \langle \textit{Identifier} \rangle \\
& & \texttt{interface} \ \langle \textit{Iface} \rangle \\
& & \texttt{body} \ \langle \textit{ModuleBody} \rangle
\end{array}$$

# SIMPLE-MODULES: Concrete Syntax

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

- Syntax for the interface of a module:

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

- Syntax for the body of a module

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

# SIMPLE-MODULES: Concrete Syntax

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

# SIMPLE-MODULES: Abstract Syntax

```
type prog = AProg of (mdecl list)*expr
```

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

# SIMPLE-MODULES: Abstract Syntax

```
type mdecl = AMod of string*interface*module_body
```

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

# SIMPLE-MODULES: Abstract Syntax

```
type interface = ASimpleInterface of vdecl list
type module_body = AModBody of vdef list
```

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

# Concrete vs Abstract Syntax

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

```
AProg
 ([AMod ("m1",
   ASimpleInterface [AValDecl ("u",IntType); AValDecl ("v", IntType)],
   AModBody [AValDef ("v",Int 33); AValDef ("u",Int 44)])],
 QualVar ("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.

```
env =
| EmptyEnv
| ExtendEnv of string*exp_val*env
| ExtendEnvRec of string*string*Ast.expr*env
| ExtendEnvMod of string*env*env
```

# 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 and Store extant at line 17

```
Environment:
[Module m2 [(b,NumVal 77)(a,NumVal 66)];
 Module m1 [(c,NumVal 55)(b,NumVal 44)(a,NumVal 33)];
 (i,NumVal 1);
 (v,NumVal 5);
 (x,NumVal 10)]
Store:
```

# Body Evaluation

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

```
from m take var
```

- We use `apply_env_qual`
- This first looks up the module `m` in the current environment, and then looks up `var` in the resulting environment.

```
let rec apply_env_qual (env:env) (mid:string) (id:string): exp_val option =
  match env with
  | EmptyEnv -> None
  | ExtendEnv (key,value,env) -> apply_env_qual env mid id
  | ExtendEnvRec(key,param,body,env) -> apply_env_qual env mid id
  | ExtendEnvMod(moduleName,bindings,env) ->
    if mid=moduleName
    then apply_env bindings id
    else apply_env_qual env  mid  id
```

# Program Evaluation

```
eval_prog (Ast.AProg(ms,e)) =
    eval_expr (add_module_definitions init_env ms) e
```

# Program Evaluation

```
eval_prog (Ast.AProg(ms,e)) =
   eval_expr (add_module_definitions init_env ms) e
```

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

# Program Evaluation

```
add_module_definition (en:env) (AModBody vdefs):env =
  snd @@ List.fold_left (fun (env,renv) (AValDef(var,decl)) ->
      let ev=eval_expr env decl
      in (ExtendEnv(var,ev,env),ExtendEnv(var,ev,renv))) (en,EmptyEnv)
    vdefs
and
  add_module_definitions (en:env) (ms:mdecl list):env =
     List.fold_left (fun curr_en (AMod(mname,minterface,mbody)) ->
      ExtendEnvMod(mname,add_module_definition curr_en mbody,curr_en) ) en
    ms
```

- `add_module_definition` evaluates each definition in the body of the module

Simple Modules

The Interpreter

The Type-Checker

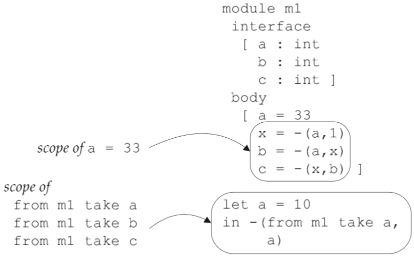
# The Type-Checker

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

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

# Typing Judgements for Module Declarations

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

- $\mathtt{m\_tenv}_2$ is the type of the list of modules $\mathtt{M}$
- $\mathtt{m\_tenv}_1$ is the type of the list of modules that $\mathtt{M}$ 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 $f : I \to J$

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

# 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

```
type tenv =
  | EmptyTEnv
  | ExtendTEnv of string*texpr*tenv
  | ExtendTEnvMod of string*tenv*tenv
```

# Type-Checking Programs

```
let rec type_of_prog (AProg (ms,e)) =
      type_of_expr (add_module_type_definitions (init_tenv
          ↪ ()) ms) e
```

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

# Type-Checking Qualified Variables

`from m take var`

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

```
type_of_expr en = function
| Int n            -> IntType
| Var id            ->
  (match apply_tenv en id with
  | None -> failwith @@ "Variable "^id^" undefined"
  | Some texp -> texp)
| QualVar(module_id,var_id) ->
  (match apply_tenv_qual en module_id var_id with
   | None -> failwith @@ "Variable "^var_id^" undefined"
   | Some texp ->  texp)
```

# 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.

```
let rec apply_tenv_qual (tenv:tenv) (id_module:string) (id:string):texpr option =
  match tenv with
  | EmptyTEnv -> None
  | ExtendTEnv (key,value,tenv1) ->
        apply_tenv_qual tenv1 id_module id
  | ExtendTEnvMod(m_name,m_type,tenv1) ->
        if id_module=m_name
        then apply_tenv m_type id
        else apply_tenv_qual tenv1 id_module id
```

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

# 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

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

# Type-Checking Modules

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

$$\frac{}{\texttt{m\_tenv} \vdash \epsilon :: \texttt{[]}}\ \textit{TModE}$$

# Type-Checking Programs

```
let rec
  type_of_prog (AProg (ms,e)) =
      type_of_expr (add_module_type_definitions (init_tenv
       ↪ ()) ms) e
and
  add_module_type_definitions tenv:(mdecl list->tenv)  = ...
and
  type_of_module_body en (AModBody vdefs):tenv = ...
```

$$\frac{\vdash \mathtt{M} :: \mathtt{m\_tenv} \quad \mathtt{m\_tenv;empty\text{-}tenv} \vdash \mathtt{e} :: \mathtt{t}}{\vdash \mathtt{M} \;\; \mathtt{e} :: \mathtt{t}}\ \mathit{TProgram}$$

# Type-Checking Programs

```
let rec
  type_of_prog (AProg (ms,e)) =
        type_of_expr (add_module_type_definitions (init_tenv ()) ms) e
and
  add_module_type_definitions tenv:(mdecl list->tenv)  = ...
and
  type_of_module_body en (AModBody vdefs):tenv = ...
```

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

# Type-Checking Programs

```
let rec
  type_of_prog (AProg (ms,e)) =
      type_of_expr (add_module_type_definitions (init_tenv
          ↪ ()) ms) e
and
  add_module_type_definitions tenv:(mdecl list->tenv)  = ...
and
  type_of_module_body en (AModBody vdefs):tenv = ...
```

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

# Type-Checking Programs

```
add_module_type_definitions tenv:(mdecl list->tenv)  = function
  | [] -> tenv
  | AMod(mname,ASimpleInterface(expected_iface),mbody)::ms ->
    let i_body = type_of_module_body tenv mbody
    in if (is_subtype i_body expected_iface)
    then add_module_type_definitions
              (ExtendTEnvMod(mname,var_decls_to_tenv expected_iface,tenv))
             ms
    else raise (Subtype_failure(mname))
and
  type_of_module_body en (AModBody vdefs):tenv =
  reverse_tenv @@ snd @@ List.fold_left (fun (env,renv) (AValDef(var,decl))  ->
      let ty = type_of_expr env decl
      in (ExtendTEnv(var, ty,env),ExtendTEnv(var, ty,renv)))  (en,EmptyTEnv) vdefs
```

$$\frac{\vdash M :: \mathtt{m\_tenv} \quad \mathtt{m\_tenv;empty\text{-}tenv} \vdash e :: t}{\vdash M\ e :: t}\ \mathit{TProgram}$$

# Subtype Checking

```
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]
```

```
let rec is_subtype (actual:tenv) (expected: vdecl list) =
  match actual, expected with
  | _,[] -> true
  | EmptyTEnv,_::_ -> false
  | ExtendTEnv(ida,tya,tenv),(AValDecl(ide,tye))::ys ->
    if ida=ide
    then tya=tye && is_subtype tenv ys
    else is_subtype tenv ((AValDecl(ide,tye))::ys)
  | _,_ -> failwith "Case not reachable"
```

# The Interpreter for SIMPLE-MODULES

- Code available in Canvas `Modules/Interpreters`
- Directory `simple-modules`
- Compile with `ocamlbuild -use-menhir checker.ml`
- Make sure the `.ocamlinit` file is in the folder of your sources
- Run `utop`

# Beyond SIMPLE-MODULES

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