# Typechecking

CS496

# Types

- Types
  - Organize data and act as classifiers.
  - Constitute a form of documentation.
  - Provide an approximation of the behavior of an expression.
- A type can state that something is a number, a list, a character, a string, a procedure, etc.
- The type of a procedure declares the types of its arguments, and when a procedure is applied to arguments of the wrong type a type error occurs.

# Types

1. Static vs dynamic typing
   - Compile time
   - Run time
2. Type checking vs type inference

# Typed Languages

- Define a set of types and when an expression `e` has type `t`, written

  `e::t`

- A type analysis step is introduced into the language-processing model.
  - It tries to assign a type to each expression in the program.
  - It reports an error if it can't.

# Examples

- In order to gain further intuition on typability, we next consider a series of examples
- For each we ask ourselves:
  - Is this expression typable?
  - If so, what should its type be?
- Let's start with

```
if 3 then 88 else 99
```

# Examples

- In order to gain further intuition on typability, we next consider a series of examples
- For each we ask ourselves:
  - Is this expression typable?
  - If so, what should its type be?
- Let's start with

```
if 3 then 88 else 99
```
reject: 3 is not a boolean

```
proc (x){ (3 x)}
```

# Examples

- In order to gain further intuition on typability, we next consider a series of examples
- For each we ask ourselves:
  - Is this expression typable?
  - If so, what should its type be?
- Let's start with

| | |
|---|---|
| `if 3 then 88 else 99` | reject: 3 is not a boolean |
| `proc (x){ (3 x)}` | reject: non-proc-val rator<br>Does the type of `x` matter here? |

# Examples

```
proc (x){ (x 3)}
```

# Examples

```
proc (x){ (x 3)}
```

- It depends on the type of `x`
- For example, if `x` is a boolean, then its not well typed.
- But if `x` has the type of a function that consumes numbers, then it is well-typed
- We accept this procedure expression as typable because there is a type for `x` that makes its body typable

# More Examples

```
proc (f){ proc (x){ (f x)} }
```

# More Examples

```
proc (f){ proc (x){ (f x)} }
```

accept

```
let x = 4 in (x 3)
```

# More Examples

| | |
|---|---|
| `proc (f){ proc (x){ (f x)} }` | accept |
| `let x = 4 in (x 3)` | reject: non-proc-val rator |
| `(proc (x){ (x 3)} 4)` | |

# More Examples

| | |
|---|---|
| `proc (f){ proc (x){ (f x)} }` | accept |
| `let x = 4 in (x 3)` | reject: non-proc-val rator |
| `(proc (x){ (x 3)} 4)` | reject: same as preceding example |

```
let x = zero?(0)
in 3-x
```

# More Examples

| | |
|---|---|
| `proc (f){ proc (x){ (f x)} }` | accept |
| `let x = 4 in (x 3)` | reject: non-proc-val rator |
| `(proc (x){ (x 3)} 4)` | reject: same as preceding example |

```
let x = zero?(0)
in 3-x
```

reject: non-integer argument to a diff-ex

# More Examples

```
(proc (x){ 3-x }
zero?(0))
```

# More Examples

```
(proc (x){ 3-x }
zero?(0))
```

reject: same as preceding example

```
let f = 3
in proc (x){ (f x)}
```

# More Examples

```
(proc (x){ 3-x }
zero?(0))
```

reject: same as preceding example

```
let f = 3
in proc (x){ (f x)}
```

reject: non-proc-val rator

```
(proc (f)proc (x){ (f x)}
3)
```

# More Examples

```
(proc (x){ 3-x }
zero?(0))
```

reject: same as preceding example

```
let f = 3
in proc (x){ (f x)}
```

reject: non-proc-val rator

```
(proc (f)proc (x){ (f x)}
3)
```

reject: same as preceding example

```
letrec f(x)= (f (x+1))
in (f 1)
```

# More Examples

```
(proc (x){ 3-x }
zero?(0))
```

reject: same as preceding example

```
let f = 3
in proc (x){ (f x)}
```

reject: non-proc-val rator

```
(proc (f)proc (x){ (f x)}
3)
```

reject: same as preceding example

```
letrec f(x)= (f (x+1))
in (f 1)
```

accept, nonterminating but safe

# Typable Expressions and Evaluation Safety

- If an expression can be assigned a type we say it is typable
- What guarantees do typable expressions give us at run-time?
  - They guarantee that evaluation (i.e. execution) is safe
- For example, that every evaluation of a variable `var`is found in the environment.
- We next give a definition of what it means for evaluation to be safe

# Evaluation Safety

Evaluation is safe if and only if for every evaluation of a(n):

1. variable *var*, the variable is bound.
2. `exp1-exp2`, the values of *exp1* and *exp2* are both num-vals.
3. `zero?(exp1)`, the value of *exp1* is a num-val.
4. `if exp1 then exp2 else exp3`, the value of *exp1* is a bool-val.
5. `(rator rand)`, the value of *rator* is a proc-val.

Evaluation of safe programs may fail: division by zero, car of the empty list, infinite loop, etc.

# Concrete Syntax of Types

$$
\begin{array}{lcl}
\langle \mathit{Type} \rangle & ::= & \texttt{int} \\
\langle \mathit{Type} \rangle & ::= & \texttt{bool} \\
\langle \mathit{Type} \rangle & ::= & \langle \mathit{Type} \rangle \ \texttt{->} \ \langle \mathit{Type} \rangle \\
\langle \mathit{Type} \rangle & ::= & (\langle \mathit{Type} \rangle)
\end{array}
$$

# Examples of Values and Their Types

- Recall that we write

  `e::t`

  if expression `e` has type `t`

Examples:

- `3::int`
- `33-22::int`
- `zero?(11)::bool`
- `proc (x){ x-11 }::(int→int)`
- `proc (x)let y = -(x,11)in x-y:: (int→int)`
- `proc (x){ if x then 11 else 22 }::(bool→int)`

# More Examples of Values and Their Types

- `proc (x){ if x then 11 else zero?(11)}` has no type.
- `proc (x){ proc (y){ if y then x else 11 } } :: int→(bool→ int)`.
- `proc (f){ if (f 3)then 11 else 22 } :: (int→bool)→int`
- `proc (f){ (f 3)} :: (int→t)→t`, for any type t.
- `proc (f){ proc (x){ (f (f x))} } :: ((t→t)→(t→t))`, for any type t.

Typed Languages

Specifying the Behavior of the Type Checker

The Language CHECKED

Typing Letrec

# Typing Rules

- What is the type of `3`?
- What is the type of `zero?(4)`?
- What is the type of `zero?(x)`?

# Typing Rules

- What is the type of `3`?
- What is the type of `zero?(4)`?
- What is the type of `zero?(x)`?
- We need to know the types of the variables in order to determine the type of an expression
- A type environment `tenv` associates types to variables
  - E.g. $\{x \leftarrow bool, y \leftarrow int\}$

# Typing Judgements

- A typing judgement is an expression of the form

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

  where

  - `tenv` is a type environment
  - `e` is an expression
  - `t` is a type expression

- A typing system consists of typing rules

$$\frac{J_1 \ldots J_n}{J}\,\texttt{rule-name}$$

  - $J_1, \ldots, J_n, J$ are typing judgements
  - When $n = 0$, the rule is also called an axiom

# Typing Derivations

- Typing rules can be composed to form typing derivations
- A typing system determines a set of derivable typing judgements, namely those that are the root of a typing derivation
- If a judgement

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

  is derivable, then we say that "e is typable with type t under typing environment tenv"

# Preliminary Summary of Notions

- Typing judgement:

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

- Typing rule:

$$\frac{J_1 \ldots J_n}{J}\,\texttt{rule-name}$$

- Typing derivation: Tree of typing judgements built from typing rules
- Derivable typing judgements: Those that are the root of a typing derivation

# Typing Rules

## Typing axioms and rules for expressions

Typing integers:

$$\frac{}{\texttt{tenv} \vdash \texttt{n} :: \texttt{int}} \; \textit{TConst}$$

Typing variables:

$$\frac{\texttt{tenv(x)=t}}{\texttt{tenv} \vdash \texttt{x} :: \texttt{t}} \; \textit{TVar}$$

# Typing Rules

Typing zero?:

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{int}}{\texttt{tenv} \vdash \texttt{zero?(e)} :: \texttt{bool}} \; \textit{TZero}$$

Typing diff:

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{int} \qquad \texttt{tenv} \vdash \texttt{e2} :: \texttt{int}}{\texttt{tenv} \vdash \texttt{e1-e2} :: \texttt{int}} \; \textit{TDiff}$$

# Typing rules – If

$$\frac{\begin{array}{l}\texttt{tenv} \vdash \texttt{e1} :: \texttt{bool} \\ \texttt{tenv} \vdash \texttt{e2} :: \texttt{t} \\ \texttt{tenv} \vdash \texttt{e3} :: \texttt{t}\end{array}}{\texttt{tenv} \vdash \texttt{if e1 then e2 else e3} :: \texttt{t}}\ \textit{TIf}$$

# Exercise Before Continuing

- Show that `if zero?(0)then 3 else 4` is typable
- For that, construct a typing derivation for the judgement

  $\texttt{empty-tenv} \vdash \texttt{if zero?(0) then 3 else 4} :: \texttt{int}$
- Note that in a typing derivation
  - Each leaf of the tree is an instance of an axiom;
  - Each internal node is an instance of a typing rule; and
  - The root of the tree is

    $\texttt{empty-tenv} \vdash \texttt{if zero?(0) then 3 else 4} :: \texttt{int}$

# Typing rules – Let

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{t1} \quad \texttt{[var=t1]tenv} \vdash \texttt{e2} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{let var=e1 in e2} :: \texttt{t2}}\ \textit{TLet}$$

# Typing rules – Proc Application

$$\frac{\texttt{tenv} \vdash \texttt{rator} :: \texttt{t1} \rightarrow \texttt{t2} \quad \texttt{tenv} \vdash \texttt{rand} :: \texttt{t1}}{\texttt{tenv} \vdash \texttt{(rator rand)} :: \texttt{t2}}\ \textit{TProcApp}$$

# Typing rules

## Attempt at typing procedures

Motivating expression: `proc (x)-(x,2)`

$$\frac{[\texttt{var = t1}]\texttt{tenv} \vdash \texttt{e} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{proc (var) \{e\}} :: \texttt{t1} \rightarrow \texttt{t2}} \; \textit{TProc}$$

# Typing rules

## Attempt at typing procedures

Motivating expression: `proc (x)-(x,2)`

$$\frac{\texttt{[var = t1]tenv} \vdash \texttt{e} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{proc (var) \{e\}} :: \texttt{t1} \rightarrow \texttt{t2}}\ \textit{TProc}$$

- Where do we obtain `t1` from?
- This specification is incomplete as it stands
- Two options:
  1. the missing type is supplied by the programmer (we choose this one for now!)
  2. the missing type is inferred from the source code

# Typing `proc`

Failed attempt:

$$\frac{[\texttt{var = t1}]\texttt{tenv} \vdash \texttt{e} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{proc (var) \{e\}} :: \texttt{t1} \rightarrow \texttt{t2}} \; \textit{TProc}$$

New typing rule:

$$\frac{[\texttt{var = t1}]\texttt{tenv} \vdash \texttt{e} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{proc (var:t1) \{e\}} :: \texttt{t1} \rightarrow \texttt{t2}} \; \textit{TProc}$$

# Summary of Typing Rules

$$\frac{}{\texttt{tenv} \vdash \texttt{n} :: \texttt{int}}\ \textit{TConst} \qquad \frac{\texttt{tenv(x)=t}}{\texttt{tenv} \vdash \texttt{x} :: \texttt{t}}\ \textit{TVar} \qquad \frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{int}}{\texttt{tenv} \vdash \texttt{zero?(e)} :: \texttt{bool}}\ \textit{TZero}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{int} \qquad \texttt{tenv} \vdash \texttt{e2} :: \texttt{int}}{\texttt{tenv} \vdash \texttt{e1-e2} :: \texttt{int}}\ \textit{TDiff}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{bool} \quad \texttt{tenv} \vdash \texttt{e2} :: \texttt{t} \quad \texttt{tenv} \vdash \texttt{e3} :: \texttt{t}}{\texttt{tenv} \vdash \texttt{if e1 then e2 else e3} :: \texttt{t}}\ \textit{TIf}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{t1} \quad \texttt{[var=t1]tenv} \vdash \texttt{e2} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{let var=e1 in e2} :: \texttt{t2}}\ \textit{TLet}$$

$$\frac{\texttt{tenv} \vdash \texttt{rator} :: \texttt{t1} \rightarrow \texttt{t2} \quad \texttt{tenv} \vdash \texttt{rand} :: \texttt{t1}}{\texttt{tenv} \vdash \texttt{(rator rand)} :: \texttt{t2}}\ \textit{TProcApp}$$

$$\frac{\texttt{[var = t1]tenv} \vdash \texttt{e} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{proc (var:t1) \{e\}} :: \texttt{t1} \rightarrow \texttt{t2}}\ \textit{TProc}$$

# The Language CHECKED

- We now introduce CHECKED
- It is based on REC except that the programmer writes
  - the type of formal parameters in procedures, and
  - the type of parameters and results in `letrec`-bound variables.

# Examples

```
proc (x:int) { x-1 }
```

```
proc (f:(bool -> int)) {
        proc (n:int) { (f zero?(n)) } }
```

# CHECKED: Concrete Syntax

- One existing production (for now) is modified as follows

$\langle \mathit{Expression} \rangle ::= \texttt{proc}\ (\langle \mathit{Identifier} \rangle : \langle \mathit{Type} \rangle)\ \{\ \langle \mathit{Expression} \rangle\ \}$

- We recall the syntax of types below:

$$
\begin{array}{lcl}
\langle \mathit{Type} \rangle & ::= & \texttt{int} \\
\langle \mathit{Type} \rangle & ::= & \texttt{bool} \\
\langle \mathit{Type} \rangle & ::= & \langle \mathit{Type} \rangle\ \texttt{->}\ \langle \mathit{Type} \rangle \\
\langle \mathit{Type} \rangle & ::= & (\langle \mathit{Type} \rangle)
\end{array}
$$

# CHECKED: Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Sub of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*texpr*expr
  | App of expr*expr
and
 texpr =
  | IntType
  | BoolType
  | FuncType of texpr*texpr
```

# Concrete vs Abstract Syntax

```
proc (f:bool -> int) {
       proc (n:int) {(f zero?(n)) } }
```

```
AProg
 (Proc ("f", FuncType (BoolType, IntType),
   Proc ("n", IntType, App (Var "f", IsZero (Var "n"))))))
```

# Implementing a Type-Checker

- We implement the following:

```
(* type_of_prog :: prog -> texpr *)
(* type_of_expr :: tenv -> expr -> texpr *)
```

- We use the specification as a guideline
- Type environments

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

## type-of-program

```
let rec type_of_prog = function
  | AProg e -> type_of_expr (init_tenv ()) e
```

`init_tenv :: unit -> tenv` returns an initial type environment

```
let init_tenv () =
     extend_tenv "x"  IntType
     @@ extend_tenv "v" IntType
     @@ extend_tenv "i"  IntType
     @@ empty_tenv ()
```

# Typing Integers

$$\frac{}{\texttt{tenv} \vdash \texttt{n} :: \texttt{int}}\ \mathit{TConst}$$

```
let rec type_of_expr en = function
  | Int n           -> IntType
```

# Typing Variable References

$$\frac{\texttt{tenv(var)=t}}{\texttt{tenv} \vdash \texttt{var} :: \texttt{t}}\ \mathit{TVar}$$

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

# Typing the `zero?` Predicate

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{int}}{\texttt{tenv} \vdash \texttt{zero?(e)} :: \texttt{bool}} \; \textit{TZero}$$

```
let rec type_of_expr en = function
  ...
  | IsZero(e) ->
    let t1 = type_of_expr en e  in
    if t1=IntType
    then BoolType
    else failwith "Zero?: argument must be int"
```

# Typing Difference

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{int} \qquad \texttt{tenv} \vdash \texttt{e2} :: \texttt{int}}{\texttt{tenv} \vdash \texttt{e1-e2} :: \texttt{int}}\ \textit{TDiff}$$

```
let rec  type_of_expr en = function
  ...
  | Add(e1, e2) | Mul(e1,e2) | Sub(e1,e2) | Div(e1,e2) ->
    let t1 = type_of_expr en e1 in
    let t2 = type_of_expr en e2  in
    if t1=IntType && t2=IntType
    then IntType
    else failwith "Add: arguments must be ints"
```

# Typing the Conditional

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{bool} \quad \texttt{tenv} \vdash \texttt{e2} :: \texttt{t} \quad \texttt{tenv} \vdash \texttt{e3} :: \texttt{t}}{\texttt{tenv} \vdash \texttt{if e1 then e2 else e3} :: \texttt{t}} \; \mathit{TIf}$$

```
let rec  type_of_expr en = function
  ...
  | ITE(e1, e2, e3)    ->
    let t1 = type_of_expr en e1
    in let t2 = type_of_expr en e2
    in let t3 = type_of_expr en e3
    in if t1=BoolType && t2=t3
    then t2
    else failwith "ITE: Type error"
```

# Typing `let`

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{t1} \qquad \texttt{[var=t1]tenv} \vdash \texttt{e2} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{let var=e1 in e2} :: \texttt{t2}}\ \textit{TLet}$$

```
let rec  type_of_expr en = function
  ...
  | Let(var, e1, e2) ->
    let t1 = type_of_expr en e1
    in type_of_expr (extend_tenv var t1 en) e2
```

# Typing Procedure Declaration

$$\frac{\texttt{[var = t1]tenv} \vdash \texttt{e} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{proc (var:t1) e} :: \texttt{t1} \rightarrow \texttt{t2}}\ \textit{TProc}$$

```
let rec  type_of_expr en = function
  ...
  | Proc(var,t1,e)      ->
    let t2= type_of_expr (extend_tenv var t1 en) e
    in FuncType(t1,t2)
```

# Typing Procedure Application

$$\frac{\texttt{tenv} \vdash \texttt{rator} :: \texttt{t1} \rightarrow \texttt{t2} \quad \texttt{tenv} \vdash \texttt{rand} :: \texttt{t1}}{\texttt{tenv} \vdash \texttt{(rator rand)} :: \texttt{t2}} \; \mathit{TProcApp}$$

```
let rec type_of_expr en = function
  ...
  | App(e1,e2)      ->
    let tl = type_of_expr en e1
    in let tr = type_of_expr en e2
    in (match tl with
    | FuncType(t1,t2) when t1=tr -> t2
    | FuncType(t1,t2) -> failwith "App: argument does
       ↪ not have correct type"
    | _ -> failwith "App: LHS must be function type")
```

# Testing CHECKED

- Code available in Canvas `Modules/Interpreters`
- Directory `checked-lang`
- Compile with `make check`
- Make sure the `.ocamlinit` file is in the folder of your sources
- Run `utop`

Typed Languages

Specifying the Behavior of the Type Checker

The Language CHECKED

Typing Letrec

# Letrec

```
letrec int double (x:int) =
                    if zero?(x)
                      then 0
                      else -((double -(x,1)), -2)
in double
```

# CHECKED: Concrete Syntax

$$\begin{aligned}\langle \textit{Expression} \rangle &::= \text{proc } (\langle \textit{Identifier} \rangle : \langle \textit{Type} \rangle)\ \langle \textit{Expression} \rangle \\ \langle \textit{Expression} \rangle &::= \text{letrec } \langle \textit{Type} \rangle\ \langle \textit{Identifier} \rangle\ (\langle \textit{Identifier} \rangle : \langle \textit{Type} \rangle) = \\ &\qquad \langle \textit{Expression} \rangle \text{ in } \langle \textit{Expression} \rangle\end{aligned}$$

# CHECKED: Abstract Syntax

```
type expr =
  ...
  | Letrec of texpr*string*string*texpr*expr*expr
```

# Abstract Syntax for `letrec`

```
letrec int double (x:int) =
                  if zero?(x)
                    then 0
                    else (double (x-1)) + 2
in double
```

```
AProg
 (Letrec (IntType, "double", "x", IntType,
   ITE (IsZero (Var "x"), Int 0,
    Add (App (Var "double", Sub (Var "x", Int 1)), Int
        ↪ 2)),
   Var "double"))
```

# Typing rule for `letrec`

```
letrec int double (x:int) =
                    if zero?(x)
                      then 0
                      else (double (x-1)) + 2
in double
```

$$\dfrac{\begin{array}{l}[\mathtt{var}{=}\mathtt{tVar}][\mathtt{f}{=}\mathtt{tVar}{\to}\mathtt{tRes}]\mathtt{tenv} \vdash \mathtt{e} :: \mathtt{tRes} \\ [\mathtt{f}{=}\mathtt{tVar}{\to}\mathtt{tRes}]\mathtt{tenv} \vdash \mathtt{body} :: \mathtt{t}\end{array}}{\mathtt{tenv} \vdash \mathtt{letrec\ tRes\ f\ (var{:}tVar)\ =\ e\ in\ body} :: \mathtt{t}}\ \textit{TRec}$$

# Typing Letrec

```
let rec type_of_expr en = function
  ...
  | Letrec(tRes,id,param,tParam,body,target) ->
    let t=type_of_expr
            (extend_tenv param tParam
            (extend_tenv id (FuncType(tParam,tRes)) en ))
            body
    in if t=tRes
       then type_of_expr
                 (extend_tenv id (FuncType(tParam, tRes)) en) target
       else failwith "LetRec: Type of recursive function does not match declaration"
```