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 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-exp

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

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

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

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

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)`

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

- 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:

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

New typing rule:

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

Summary of Typing Rules

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

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

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

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

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

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

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

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

- We recall the syntax of types below:

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

CHECKED: Abstract Syntax

```
(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var symbol?))
  (let-exp
    (var symbol?)
    (exp1 expression?)
    (body expression?))
```

The new variants for type annotated procs and letrec

```
(proc-exp
  (var symbol?)
  (type type?)
  (body expression?))
(call-exp
  (rator expression?)
  (rand expression?))
```

The new variants for type annotated procs and letrec

```
(proc-exp
  (var symbol?)
  (type type?)
  (body expression?))
(call-exp
  (rator expression?)
  (rand expression?))
```

CHECKED: Abstract Syntax of Types

```
(define-datatype type type?
  (int-type)
  (bool-type)
  (proc-type
       (src type?)
       (tgt type?)))
```

Concrete vs Abstract Syntax

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

```
(a-program
 (proc-exp
  'f
  (proc-type (bool-type) (int-type))
  (proc-exp 'n
            (int-type)
            (call-exp (var-exp 'f) (zero?-exp (var-exp
    'n))))))
```

Implementing a Type-Checker

- We implement the following:

```
;; type-of-program :: program -> type
;; type-of :: {exp,typeEnv} -> type
```

- We use the specification as a guideline
- Type environments

```
(define-datatype type-environment type-environment?
  (empty-tenv-record)
  (extended-tenv-record
    (sym symbol?)
    (type type?)
    (tenv type-environment?)))
```

Implementing a Type-Checker

- We make use of the following auxiliary function:

```
;; check-equal-type! :: {type,type,exp} -> unspecified
(define check-equal-type!
  (lambda (ty1 ty2 exp)
    (when (not (equal? ty1 ty2))
      (report-unequal-types ty1 ty2 exp))))
```

  - `report-unequal-types` simply prints an error message
  - `when` is an `if` without an `else`; returns `#<void>` if the condition is false

```
;; type-of-program :: program -> type
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1) (type-of exp1 (init-tenv))))))
```

`(init-tenv)` is the type environment for the initial environment

```
(define init-tenv
    (lambda ()
      (extend-tenv 'x (int-type)
        (extend-tenv 'v (int-type)
          (extend-tenv 'i (int-type)
            (empty-tenv))))))
```

Typing Integers

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

```
(define type-of
  (lambda (exp tenv)
    (cases expression exp
      (const-exp (n) (int-type)))))
```

Typing Variable References

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

```
(define type-of
  (lambda (exp tenv)
    (cases expression exp
      ...
     (var-exp (var) (apply-tenv tenv var)))))
```

Typing the `zero?` Predicate

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

```
(define type-of
   (lambda (exp tenv)
     (cases expression exp
       ...
       (zero?-exp (e)
         (let ((ty (type-of e tenv)))
           (check-equal-type! ty (int-type) e)
           (bool-type))))))
```

Typing Difference

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

```
(define type-of
   (lambda (exp tenv)
     (cases expression exp
       ...
       (diff-exp (e1 e2)
         (let ((ty1 (type-of e1 tenv))
               (ty2 (type-of e2 tenv)))
           (check-equal-type! ty1 (int-type) e1)
           (check-equal-type! ty2 (int-type) e2)
           (int-type))))))
```

Typing `let`

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

```
(define type-of
   (lambda (exp tenv)
     (cases expression exp
       ...
       (let-exp (var e1 e2)
         (let ((e1-type (type-of e1 tenv)))
           (type-of e2
             (extend-tenv var e1-type tenv))))))))
```

Typing the Conditional

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

```scheme
(define type-of
   (lambda (exp tenv)
     (cases expression exp
       ...
       (if-exp (e1 e2 e3)
         (let ((ty1 (type-of e1 tenv))
               (ty2 (type-of e2 tenv))
               (ty3 (type-of e3 tenv)))
           (check-equal-type! ty1 (bool-type) e1)
           (check-equal-type! ty2 ty3 exp)
           ty2)))))
```

Typing Procedure Declaration

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

```
(define type-of
 (lambda (exp tenv)
   (cases expression exp
     ...
     (proc-exp (var t1 e)
       (let ((t2 (type-of e
                          (extend-tenv var t1 tenv))))
         (proc-type t1 t2))))))
```

Typing Procedure Application

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

```
(define type-of
 (lambda (exp tenv)
   (cases expression exp
     ...
     (call-exp (rator rand)
       (let ((rator-type (type-of rator tenv))
             (rand-type  (type-of rand tenv)))
         (cases type rator-type
           (proc-type (arg-type result-type)
             (begin
               (check-equal-type! arg-type rand-type rand)
               result-type))
           (else
             (report-rator-not-a-proc-type rator-type
    rator))))))))
```

Testing CHECKED

- Code available from `http://www.eopl3.com`
- Directory `chapter7/checked`
- Open `top.scm` in Racket
- There are a number of tests in `tests.scm`
- You can type-check them using `check-one`. Eg.

```
> (check-one 'apply-a-proc-2-typed)
'int
```

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

$$\langle \textit{Expression} \rangle ::= \text{proc } (\langle \textit{Identifier} \rangle : \langle \textit{Type} \rangle)\ \langle \textit{Expression} \rangle$$

$$\langle \textit{Expression} \rangle ::= \text{letrec } \langle \textit{Type} \rangle\ \langle \textit{Identifier} \rangle\ (\langle \textit{Identifier} \rangle : \langle \textit{Type} \rangle) = \langle \textit{Expression} \rangle \text{ in } \langle \textit{Expression} \rangle$$

CHECKED: Abstract Syntax

```
(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var symbol?))
  (let-exp
    (var symbol?)
    (exp1 expression?)
    (body expression?))
```

The new variant for type annotated letrec

```
(proc-exp
  (var symbol?)
  (type type?)
  (body expression?))
(call-exp
  (rator expression?)
  (rand expression?))
(letrec-exp
  (p-result-type type?)
  (p-name symbol?)
  (b-var symbol?)
  (p-var-type type?)
  (p-body expression?)
  (letrec-body expression?))
```

The new variant for type annotated letrec

```
(proc-exp
  (var symbol?)
  (type type?)
  (body expression?))
(call-exp
  (rator expression?)
  (rand expression?))
(letrec-exp
  (p-result-type type?)
  (p-name symbol?)
  (b-var symbol?)
  (p-var-type type?)
  (p-body expression?)
  (letrec-body expression?))
```

Abstract Syntax for `letrec`

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

```
(a-program
 (letrec-exp
  (int-type)
  'double
  'x
  (int-type)
  (if-exp
   (zero?-exp (var-exp 'x))
   (const-exp 0)
   (diff-exp (call-exp (var-exp 'double) (diff-exp (var-exp
    'x) (const-exp 1))) (const-exp -2)))
  (var-exp 'double)))
```

Typing rule for `letrec`

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

$$\frac{[\texttt{var}=\texttt{tVar}][\texttt{f}=\texttt{tVar}\rightarrow\texttt{tRes}]\texttt{tenv} \vdash \texttt{e} :: \texttt{tRes} \qquad [\texttt{f}=\texttt{tVar}\rightarrow\texttt{tRes}]\texttt{tenv} \vdash \texttt{body} :: \texttt{t}}{\texttt{tenv} \vdash \texttt{letrec tRes f (var:tVar) = e in body} :: \texttt{t}}\ \textit{TRec}$$

Typing Letrec

```
(define type-of
  (lambda (exp tenv)
    (cases expression exp
     ...
     (letrec-exp (p-result-type p-name b-var b-var-type
    p-body letrec-body)
       (let ((tenv-for-letrec-body
               (extend-tenv p-name
                 (proc-type b-var-type p-result-type)
                 tenv)))
         (let ((p-body-type
                 (type-of p-body
                   (extend-tenv b-var b-var-type
                     tenv-for-letrec-body))))
           (check-equal-type!
             p-body-type p-result-type p-body)
           (type-of letrec-body tenv-for-letrec-body)))))))
```