Chapter 8.
Type System

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

Review: `F-` Language

- **We defined the syntax and semantics of the following language, which we named `F-`**
  - This is the version without anonymous / recursive function

$$
\begin{aligned}
E \rightarrow\ & n \\
\mid\ & \mathbf{true} \\
\mid\ & \mathbf{false} \\
\mid\ & x \\
\mid\ & E + E \\
\mid\ & E < E \\
\mid\ & \mathbf{if}\ E\ \mathbf{then}\ E\ \mathbf{else}\ E \\
\mid\ & \mathbf{let}\ x = E\ \mathbf{in}\ E \\
\mid\ & \mathbf{let}\ f\ x = E\ \mathbf{in}\ E \\
\mid\ & E\ \ E
\end{aligned}
$$

$$\frac{}{\rho \vdash n \Downarrow n} \qquad \frac{}{\rho \vdash x \Downarrow \rho(x)}$$

$$\frac{\rho \vdash e_1 \Downarrow n_1 \quad \rho \vdash e_2 \Downarrow n_2}{\rho \vdash e_1 + e_2 \Downarrow n_1 + n_2}$$

$$\cdots$$

Review: F- Interpreter

- **We have also implemented an interpreter that runs the program according to the semantics definition**
  - *Execution of program* meant the *evaluation of expression* in **F-**

```
jschoi@cspro2:~/Lab3$ cd FMinus/
jschoi@cspro2:~/Lab3/FMinus$ ls
FMinus.fsproj  src  testcase
jschoi@cspro2:~/Lab3/FMinus$ ls src
AST.fs  FMinus.fs  Lexer.fsl  Main.fs  Parser.fsy  Types.fs
```

```
let rec evalExp (exp: Exp) (env: Env) : Val =
  ...
```

Program Error (Bugs)

- **Recall that the semantics is not defined for certain `F`-programs that are syntactically valid**
  - Intuitively, such cases correspond to program errors (bugs)
  - Ex) Type mismatch, use of unbound variable, division-by-zero
  - So far, such errors were caught at runtime (by raising exception)
- **In real-world language, there can be other various kinds of errors (bugs) as well**
  - Ex) Buffer overflow, dangling pointer, uninitialized data use, …
- **Sometimes, a program can be problematic even if its semantics is well defined**
  - Ex) Logical error, memory leak, …

Static Analysis

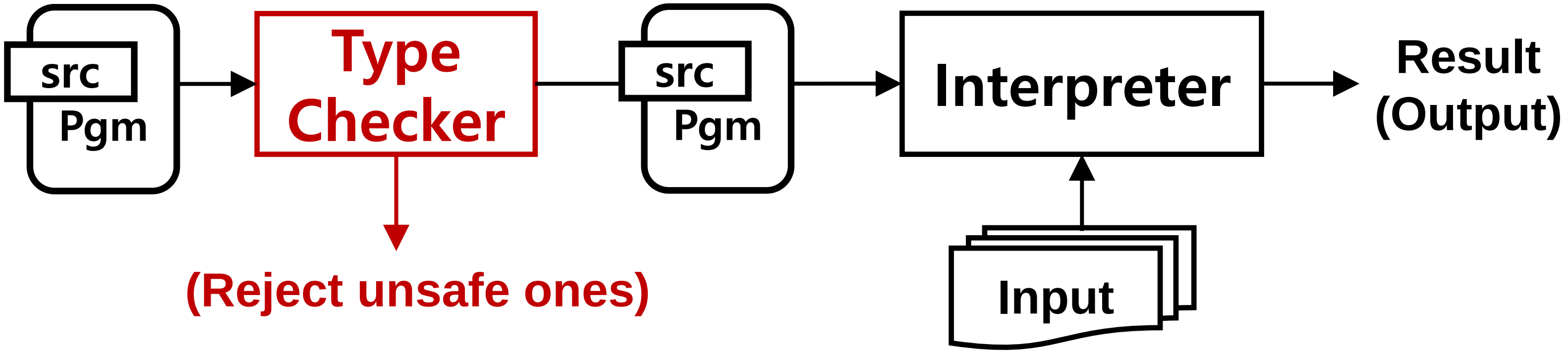
- **Everyone knows that bugs are prevalent and important**
  - Programmers cannot be free from making mistakes
  - How should we deal with these bugs?
- **How about developing an automated technique that can detect bugs before the runtime?**
  - **Automated**: analyzed by a program, not with human effort
  - **Before runtime**: to prevent it from causing a serious problem while running in the field
- **Such a technique (or the tool/program for it) is called static analysis (static analyzer)**
  - Recall that ***static*** means *"deciding before the runtime"* in PL

Bad News

- **It is known that perfect static analyzer cannot exist**
  - Writing a perfect program analysis algorithm is proven to be impossible (**"undecidable problem"**)
  - Cf. Halting problem in **Automata Theory**
- **Here, perfect means sound and complete**
  - **Sound**: program with error is always rejected / never misses an error / if a program passes, it is guaranteed to be error-free
  - **Complete**: program without error is always accepted / never rejects a safe program / if rejected, there must be an error
- **We must give up on either soundness or completeness**
  - Sometimes, we even give up both of them
  - Still, such static analyzers (with approximation) can be useful

Static Type System

- **Static type system is one of the most primitive and popular form of static analyzer**
  - Its goal is to **automatically** detect type errors **before runtime**
  - Usually equipped as a part of compiler or interpreter
- **Which side should we choose: sound or complete?**
  - **`F#`** adopts a sound (but incomplete) type system
  - In this course, we will also discuss a **sound** type system for **`F-`**
  - Cf. Type system of **`C/C++`** is neither sound nor complete



What is Type?

- **Type is a set of values**
  - Or it can be thought as an *abstraction* of a value
  - `bool` : $\{ true, false \}$
  - `int` : $\{ \ldots, -2, -1, 0, 1, 2, \ldots \}$
  - `int -> int` : Set of functions that take in `int` and return `int`
    - (Ex in **F#**) `let incr : int -> int = fun x -> x + 1`
  - `'a -> 'a` : Set of functions that take in an arbitrary type `'a` and return the same type
    - (Ex in **F#**) `let identity : 'a -> 'a = fun x -> x`

Defining Types

- **We can use inference rules to define the set of types (T) for our `F-` language as follow**
  - Let's use τ to denote type variable ($\tau \in TyVar = String$)
    - To distinguish with program variables, we will use names that start with `'` symbol
  - Ex) **`bool`**, **`int`**, **`'a`**, **`'b`**, …
  - Ex) **`bool -> int`**, **`int -> (int -> bool)`**, **`'a -> int`**, …

$$\frac{}{\text{bool} \in T} \qquad \frac{}{\text{int} \in T} \qquad \frac{}{\tau \in T}$$

$$\frac{t_1 \in T \quad t_2 \in T}{t_1 \to t_2 \in T}$$

Type Environment

- **Before we design the type system for `F-` language, let's define type environment**
  - Type environment is a **mapping from variable to type**
    - Ex) $\{\, x \mapsto \mathrm{int}, y \mapsto \mathrm{bool}\,, z \mapsto {}'\mathrm{a}\}$
  - Let's use $\boldsymbol{\Gamma}$ to denote type environment ($\boldsymbol{\Gamma} \in \boldsymbol{TyEnv} = \boldsymbol{Var} \to \boldsymbol{T}$)
  - Cf. In the semantics definition, environment was a mapping from variable to value ($\boldsymbol{\rho} \in \boldsymbol{Env} = \boldsymbol{Var} \to \boldsymbol{Val}$)

F- Language: Typing Rule

- **Next, we define relation $\boldsymbol{\Gamma \vdash e : t}$**
  - Meaning: *"Given type environment $\boldsymbol{\Gamma}$, type of $\mathbf{e}$ must be $\boldsymbol{t}$"*
  - In other words, type of $\mathbf{e}$ is $\boldsymbol{t}$ if $\boldsymbol{\Gamma \vdash e : t}$ (it's **not** if and only if)
  - Cf. In semantics definition, we wrote $\boldsymbol{\rho \vdash e \Downarrow v}$, which meant *"Given environment $\boldsymbol{\rho}$, expression $\mathbf{e}$ is evaluated into value $\boldsymbol{v}$"*

$$\frac{}{\Gamma \vdash n : \mathrm{int}} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathrm{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathrm{bool}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{int} \quad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma \vdash e_1 + e_2 : \mathrm{int}} \qquad \frac{\Gamma \vdash e_1 : \mathrm{int} \quad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma \vdash e_1 < e_2 : \mathrm{bool}}$$

`F-` Language: Typing Rule

- **Next, we define relation $\Gamma \vdash e : t$ (continued)**
  - For **`if-then-else`** expression, both e_2 and e_3 must have the same type (t) in order to prove $\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : t$
  - Ex) Consider program $\boldsymbol{e}$ = "**`if true then 1 else false`**":
    - We can prove $\phi \vdash \boldsymbol{e} \Downarrow \mathbf{1}$ (i.e., execution result of $\boldsymbol{e}$ is $\mathbf{1}$)
    - But we can't prove $\phi \vdash \boldsymbol{e} : \textbf{int}$ (typing rule does not accept it)

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma[x \mapsto \mathrm{t}_1] \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t_2}$$

Derivation Tree: Exercise

- **For program e, if we can draw a derivation tree that proves $\phi \vdash e : t$, then our type system accepts e**
  - It implies that this program is free from type error
- **Fill in the derivation tree for the program below**

$$\frac{\qquad}{\phi \vdash \textbf{let}\, x = 3 + 2\ \textbf{in}\ (x < 7) : \text{bool}}$$

Derivation Tree: Solution

- **If you are confused, remember that derivation tree is simply an application of inference rules**
  - The whole program has the form of `let x = e1 in e2`, so we should apply the following inference rule
  - Instantiate $\boldsymbol{\Gamma}$ with $\boldsymbol{\phi}$, $\boldsymbol{e_1}$ with $\mathbf{3+2}$, and $\boldsymbol{e_2}$ with $\boldsymbol{x<7}$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma[x \mapsto \mathrm{t}_1] \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 : t_2}$$

$$\frac{}{\phi \vdash \textbf{let}\, x = 3 + 2 \,\textbf{in}\, (x < 7) : \mathrm{bool}}$$

Derivation Tree: Solution

- **If you are confused, remember that derivation tree is simply an application of inference rules**
  - **Left subtree** must prove $\boldsymbol{\phi} \vdash \mathbf{3+2} : \mathbf{int}$
  - **Right subtree** must prove $\{\boldsymbol{x} \mapsto \mathbf{int}\} \vdash \boldsymbol{x} < \mathbf{7} : \mathbf{bool}$

$$\dfrac{\dfrac{}{\phi \vdash 3 + 2 : \text{int}} \qquad \dfrac{}{\phi[x \mapsto \text{int}] \vdash x < 7 : \text{bool}}}{\phi \vdash \textbf{let}\, x = 3 + 2\; \textbf{in}\; (x < 7) : \text{bool}}$$

Derivation Tree: Solution

- **The full derivation tree is as follow**
  - Note which inference rule is applied to each part of the subtree

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\boxed{\dfrac{\dfrac{}{\phi \vdash 3 : \text{int}} \qquad \dfrac{}{\phi \vdash 2 : \text{int}}}{\phi \vdash 3 + 2 : \text{int}}} \qquad \dfrac{\boxed{\dfrac{}{\{x \mapsto \text{int}\} \vdash x : \text{int}}} \qquad \dfrac{}{\{x \mapsto \text{int}\} \vdash 7 : \text{int}}}{\{x \mapsto \text{int}\} \vdash x < 7 : \text{bool}}}{\phi \vdash \textbf{let}\ x = 3 + 2\ \textbf{in}\ (x < 7) : \text{bool}}$$

`F-` Language: Typing Rule

- **Next, we define relation $\boldsymbol{\Gamma \vdash e : t}$ (continued)**
  - Consider **`let f x = e1 in e2`** (function definition)
    - Assume that **`e1`** has type $\boldsymbol{t_r}$ when argument **`x`** has type $\boldsymbol{t_a}$
    - Then, the type of function **`f`** is $\boldsymbol{t_a \rightarrow t_r}$

$$\frac{\Gamma[x \mapsto t_a] \vdash e_1 : t_r \qquad \Gamma[f \mapsto (t_a \rightarrow t_r)] \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\, f\ x = e_1\ \textbf{in}\, e_2 : t_2}$$

  - Consider **`e1 e2`** (function application)
    - If function **`e1`** has type $\boldsymbol{t_a \rightarrow t_r}$ and argument **`e2`** has type $\boldsymbol{t_a}$, the type of function call result is $\boldsymbol{t_r}$

$$\frac{\Gamma \vdash e_1 : t_a \rightarrow t_r \qquad \Gamma \vdash e_2 : t_a}{\Gamma \vdash e_1\ \ e_2 : t_r}$$

Derivation Tree: Another Exercise

- Let's prove $\phi \vdash e : t$ for the following program
- Fill in the derivation tree below

$$\frac{}{\phi \vdash \textbf{let}\ f\ x = x < 1\ \textbf{in}\ f\ 5 : \text{bool}}$$

Derivation Tree: Solution

- **Again, just choose and apply proper inference rule**
  - Apply the following inference rule, while instantiating $\boldsymbol{\Gamma}$ with $\boldsymbol{\phi}$, $\boldsymbol{e_1}$ with $\boldsymbol{x < 1}$, and $\boldsymbol{e_2}$ with $\boldsymbol{f\ 5}$
  - Also, we (intuitively) know that $\boldsymbol{t_a}$ must be $\mathbf{int}$ and $\boldsymbol{t_r}$ is $\mathbf{bool}$

$$\frac{\Gamma[x \mapsto t_a] \vdash e_1 : t_r \qquad \Gamma[f \mapsto (t_a \rightarrow t_r)] \vdash e_2 : t_2}{\Gamma \vdash \mathbf{let}\ f\ x = e_1\ \mathbf{in}\ e_2 : t_2}$$

$$\frac{}{\phi \vdash \mathbf{let}\ f\ x = x < 1\ \mathbf{in}\ f\ 5 : \mathrm{bool}}$$

Derivation Tree: Solution

- **The full derivation tree is as follow**
  - Note which inference rule is applied to each part of the subtree

$$\boxed{\frac{}{\{f \mapsto (\text{int} \to \text{bool})\} \vdash f : \text{int} \to \text{bool}}} \qquad \boxed{\frac{}{\{f \mapsto (\text{int} \to \text{bool})\} \vdash 5 : \text{int}}}$$

$$\frac{\dfrac{\dfrac{}{\{x \mapsto \text{int}\} \vdash x : \text{int}} \quad \dfrac{}{\{x \mapsto \text{int}\} \vdash 1 : \text{int}}}{\{x \mapsto \text{int}\} \vdash x < 1 : \text{bool}} \qquad \dfrac{\quad \bigcirc \quad \bigcirc \quad}{\{f \mapsto (\text{int} \to \text{bool})\} \vdash f\ 5 : \text{bool}}}{\phi \vdash \textbf{let}\ f\ x = x < 1\ \textbf{in}\ f\ 5 : \text{bool}}$$

Observation

- **Note that for certain program (e), there can be multiple types (t) such that $\phi \vdash e : t$ holds**
  - In other words, the type may not be decided uniquely
  - Consider "**`let f x = x in f`**" as example: following instances of $\phi \vdash e : t$ are all provable (i.e., we can draw derivation trees)

$$\frac{\cdots}{\phi \vdash \textbf{let}\ f\ x = x\ \textbf{in}\ f : \text{bool} \to \text{bool}} \qquad \frac{\cdots}{\phi \vdash \textbf{let}\ f\ x = x\ \textbf{in}\ f : \text{int} \to \text{int}}$$

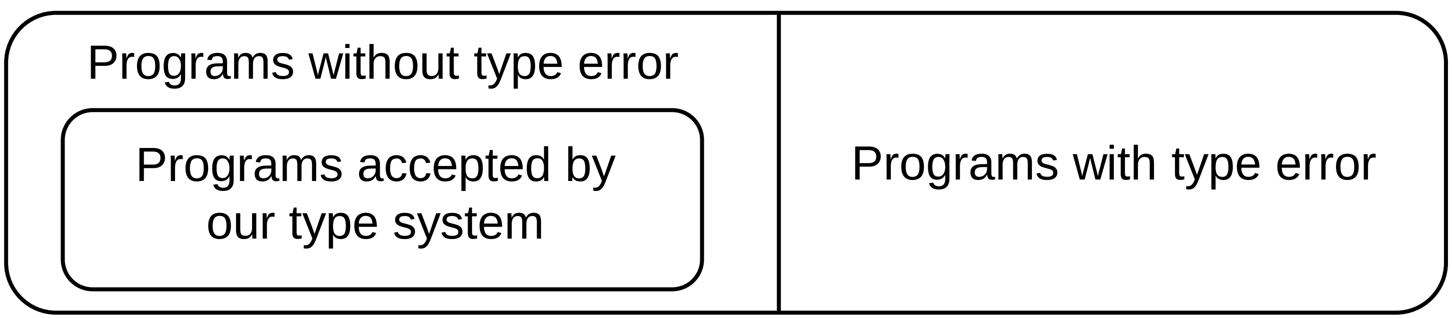
$$\frac{\cdots}{\phi \vdash \textbf{let}\ f\ x = x\ \textbf{in}\ f : (\text{int} \to \text{bool}) \to (\text{int} \to \text{bool})}$$

$$\frac{\cdots}{\phi \vdash \textbf{let}\ f\ x = x\ \textbf{in}\ f : {}'\text{a} \to {}'\text{a}}$$

Soundness of Type System

- **Recall that our goal was to design a sound but incomplete type system for `F-` language**
- **The soundness property can be described as follow**
  - If $\boldsymbol{\phi} \vdash \boldsymbol{e} : \boldsymbol{t}$ holds, then program $\boldsymbol{e}$ is free from type error
  - Also, if this program terminates and outputs $\boldsymbol{v}$ as result, the type of $\boldsymbol{v}$ is $\boldsymbol{t}$ (note that $\boldsymbol{\phi} \vdash \boldsymbol{e} : \boldsymbol{t}$ does not guarantee the termination)
- **We can even prove this (but will not in this course)***

Diagram of program set

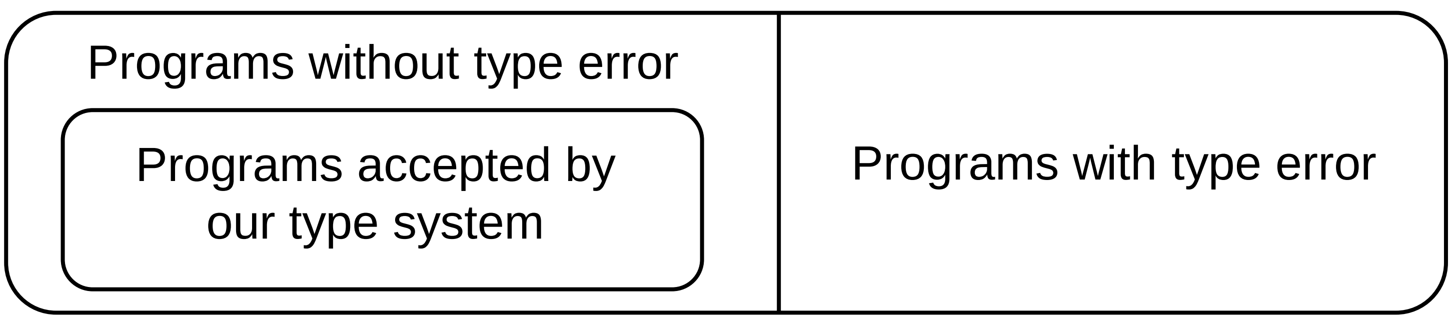


* See "*Types and Programming Languages*" by B. Pierce for details

Incompleteness of Type System

- **There can be a program that our type system does not accept, even if it does not have any type error**
  - In other words, there exists program $\boldsymbol{e}$ such that $\boldsymbol{\phi} \vdash \boldsymbol{e} \Downarrow \boldsymbol{v}$ holds for some $\boldsymbol{v}$ but $\boldsymbol{\phi} \vdash \boldsymbol{e} : \boldsymbol{t}$ does not hold for any $\boldsymbol{t}$
  - Ex) **`if true then 1 else false`**
  - Ex) **`let f x = x in if (f true) then (f 1) else 2`**
    - Our current type system does not support such polymorphism: we will briefly discuss this issue later

Diagram of program set



Derivation Tree: Why Fail?

- **`f` cannot be $\mathrm{int} \rightarrow \mathrm{int}$ and $\mathrm{bool} \rightarrow \mathrm{bool}$ at the same time**
  - Retaining `f` as $'\mathtt{a} \rightarrow {'\mathtt{a}}$ type does not solve this problem, too

Fails!

$$\frac{\overline{\{x \mapsto \mathrm{bool}\} \vdash x : \mathrm{bool}} \qquad \overline{\{f \mapsto (\mathrm{bool} \rightarrow \mathrm{bool})\} \vdash \textbf{if}\,(f\ true)\ \textbf{then}\ (f\ 1)\ \textbf{else}\ 2 : \mathrm{int}?}}{\phi \vdash \textbf{let}\ f\ x = x\ \textbf{in}\ \ \textbf{if}\,(f\ true)\ \textbf{then}\ (f\ 1)\ \textbf{else}\ 2 : \mathrm{int}?}$$

Fails!

$$\frac{\overline{\{x \mapsto \mathrm{int}\} \vdash x : \mathrm{int}} \qquad \overline{\{f \mapsto (\mathrm{int} \rightarrow \mathrm{int})\} \vdash \textbf{if}\,(f\ true)\ \textbf{then}\ (f\ 1)\ \textbf{else}\ 2 : \mathrm{int}?}}{\phi \vdash \textbf{let}\ f\ x = x\ \textbf{in}\ \ \textbf{if}\,(f\ true)\ \textbf{then}\ (f\ 1)\ \textbf{else}\ 2 : \mathrm{int}?}$$

Implementing Type System

- **The typing rules ($\Gamma \vdash e : t$) that we have discussed so far is specification of our type system**
  - Given program e, if there exists some t such that $\phi \vdash e : t$ holds, our type system must accept e
  - If we such t does not exist, our type system will not accept e
- **Now, let's think about how to actually implement it**
  - How should we write the code for this type system?

Review: Interpreter

- **When we were writing `F-` interpreter, semantics could be easily implemented with recursion**
  - We could directly translate the definition of $\boldsymbol{\rho} \vdash \boldsymbol{e} \Downarrow \boldsymbol{v}$ into code, as shown in the example below
  - Can we do the same thing for type system?

$$\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{\rho \vdash \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \Downarrow v_2}$$

```
let rec evalExp (exp: Exp) (env: Env) : Val =
  match exp with
  | LetIn (x, e1, e2) ->
    let v1 = evalExp e1 env
    evalExp e2 (Map.add x v1 env)
  | ...
```

Adaptation to Type System

- **You may think we can do the same thing**
  - It will work for most cases, as shown in the case below
  - Note that the code below looks similar to the code of interpreter

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma[x \mapsto \mathrm{t}_1] \vdash e_2 : t_2}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : t_2}$$

```
let rec typeOf (exp: Exp) (tenv: TyEnv) : Typ =
  match exp with
  | LetIn (x, e1, e2) ->
    let t1 = typeOf e1 tenv
    typeOf e2 (Map.add x t1 tenv)
  | ...
```

Challenge

- **In the typing rule below, we cannot decide the argument type (t_a) by using recursion**
  - When drawing derivation tree, you must have used *intuition* to figure it out; but how should the computers do that?

$$\frac{\Gamma[x \mapsto t_a] \vdash e_1 : t_r \qquad \Gamma[f \mapsto (t_a \to t_r)] \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\ f\ x = e_1\ \textbf{in}\ e_2 : t_2}$$

```
let rec typeOf (exp: Exp) (tenv: TyEnv) : Typ =
  match exp with
  | LetFunIn (f, x, e1, e2) ->
    let ta = ??? // What should we do here?
    let tr = typeOf e1 (Map.add x ta tenv)
    typeOf e2 (Map.add f (ta → tr) tenv)
  | ...
```

Manual vs. Automatic

- One possible solution is to enforce the programmers to write the argument type ("`let f (x: int) = ...`")
- Otherwise, we need an algorithm to automatically infer the types (continued in the next chapter)

$$\frac{\Gamma[x \mapsto t_a] \vdash e_1 : t_r \qquad \Gamma[f \mapsto (t_a \to t_r)] \vdash e_2 : t_2}{\Gamma \vdash \mathbf{let}\ f\ x = e_1\ \mathbf{in}\ e_2 : t_2}$$

```
let rec typeOf (exp: Exp) (tenv: TyEnv) : Typ =
  match exp with
  | LetFunIn (f, x, e1, e2) ->
    let ta = ??? // What should we do here?
    let tr = typeOf e1 (Map.add x ta tenv)
    typeOf e2 (Map.add f (ta → tr) tenv)
  | ...
```

More Exercises

- **Consider the extended version of `F-` language that supports recursive function and anonymous function**
  - What should be the typing rules for the following cases?
  - And should we fix the typing rule of function application (**`e1 e2`**)?

$$\frac{}{\Gamma \vdash \textbf{fun}\ x \rightarrow e\ :}$$

$$\frac{}{\Gamma \vdash \rho \vdash \textbf{let rec}\ f\ x = e_1\ \textbf{in}\ e_2\ :}$$

- **Also, draw derivation trees for various examples**
  - You can find more examples in our reference material (https://prl.korea.ac.kr/courses/cose212/2023/pl-book.pdf)
  - But note that the language can be slightly different from ours