Final Exam

COSE212: Programming Languages

2025 Fall

Instructor: Jihyeok Park

December 17, 2025. 18:30-21:00

- **If you are not good at English, please write your answers in Korean.**
  (영어가 익숙하지 않은 경우, 답안을 한글로 작성해 주세요.)
- **Write answers in good handwriting.**
  If we cannot recognize your answers, you will not get any points.
  (글씨를 알아보기 힘들면 점수를 드릴 수 없습니다. 답안을 읽기 좋게 작성해주세요.)
- **Write your answers in the boxes provided.**
  (답안을 제공된 박스 안에 작성해 주세요.)
- **There are** 10 **pages and** 10 **questions.**
  (시험은 10 장으로 총 10 문제로 구성되어 있습니다.)
- **Syntax, semantics, and typing rules of languages are given in Appendix.**
  (언어의 문법, 의미, 타입 규칙은 부록에서 참조할 수 있습니다.)

Student ID	
Student Name	

Question:	1	2	3	4	5	6	7	8	9	10	Total
Points:	10	15	10	5	8	7	10	15	15	5	100
Score:											

1. 10 points [☆☆☆] The following sentences explain basic concepts of programming languages. Fill in the blanks with the following terms (**2 points per blank**):

algebraic	continuation	equivalence	intersection	product
recursive	subtype	sum	type argument	type checking
type constraint	type inference	type instantiation	type variable	union

- A(n) [ ] is a representation of the execution state of a program at a certain point, encapsulating the rest of the computation to be performed.
- A(n) [ ] type is a compound type that contains a value for each of its component types, corresponding to the Cartesian product in set theory.
- A(n) [ ] relation is a binary relation between types, denoted as $S <: T$, meaning that any term of type S can be safely used in a context where a term of type T is expected.
- Parametric polymorphism enables writing generic entities (e.g., functions, data types) by introducing [ ] that can be instantiated with different types.
- [ ] is the process by which a compiler automatically deduces the types of expressions in a program from context, without requiring explicit type annotations from the programmer.

2. 15 points Consider a language KFAE defined with the following syntax and small-step operational semantics. It supports **first-class functions** and **first-class continuations**.

$$
\begin{array}{lll}
\text{Expressions} & \mathbb{E} \ni e & ::= n \mid e \texttt{ + } e \mid e \texttt{ * } e \mid x \mid \lambda x.e \mid e(e) \mid \texttt{vcc } x;\ e \\
\text{Values} & \mathbb{V} \ni v & ::= n \mid \langle \lambda x.e, \sigma \rangle \mid \langle \kappa \mid\mid s \rangle \\
\text{Continuations} & \mathbb{K} \ni \kappa & ::= \square \mid (\sigma \vdash e) :: \kappa \mid (+) :: \kappa \mid (\times) :: \kappa \mid (@) :: \kappa \\
\text{Environments} & & \sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} \\
\text{Value Stacks} & \mathbb{S} \ni s & ::= \blacksquare \mid v :: s
\end{array}
$$

$$\boxed{\langle \kappa \mid\mid s \rangle \to \langle \kappa \mid\mid s \rangle}$$

$$
\begin{array}{lcl}
\langle (\sigma \vdash n) :: \kappa \mid\mid s \rangle & \to & \langle \kappa \mid\mid n :: s \rangle \\
\langle (\sigma \vdash e_1 \texttt{ + } e_2) :: \kappa \mid\mid s \rangle & \to & \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (+) :: \kappa \mid\mid s \rangle \\
\langle (+) :: \kappa \mid\mid n_2 :: n_1 :: s \rangle & \to & \langle \kappa \mid\mid (n_1 + n_2) :: s \rangle \\
\langle (\sigma \vdash e_1 \texttt{ * } e_2) :: \kappa \mid\mid s \rangle & \to & \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (\times) :: \kappa \mid\mid s \rangle \\
\langle (\times) :: \kappa \mid\mid n_2 :: n_1 :: s \rangle & \to & \langle \kappa \mid\mid (n_1 \times n_2) :: s \rangle \\
\langle (\sigma \vdash x) :: \kappa \mid\mid s \rangle & \to & \langle \kappa \mid\mid \sigma(x) :: s \rangle \\
\langle (\sigma \vdash \lambda x.e) :: \kappa \mid\mid s \rangle & \to & \langle \kappa \mid\mid \langle \lambda x.e, \sigma \rangle :: s \rangle \\
\langle (\sigma \vdash e_1(e_2)) :: \kappa \mid\mid s \rangle & \to & \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (@) :: \kappa \mid\mid s \rangle \\
\langle (@) :: \kappa \mid\mid v_2 :: \langle \lambda x.e, \sigma \rangle :: s \rangle & \to & \langle (\sigma[x \mapsto v_2] \vdash e) :: \kappa \mid\mid s \rangle \\
\langle (@) :: \kappa \mid\mid v_2 :: \langle \kappa' \mid\mid s' \rangle :: s \rangle & \to & \langle \kappa' \mid\mid v_2 :: s' \rangle \\
\langle (\sigma \vdash \texttt{vcc } x;\ e) :: \kappa \mid\mid s \rangle & \to & \langle (\sigma[x \mapsto \langle \kappa \mid\mid s \rangle] \vdash e) :: \kappa \mid\mid s \rangle
\end{array}
$$

The desugaring function $\mathcal{D}[\![-]\!]$ is defined as follows, and recursive cases are omitted.

$$\mathcal{D}[\![\texttt{val } x \texttt{ = } e_1\texttt{; } e_2]\!] = (\lambda x.\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e_1]\!])$$

(a) 10 points [★☆☆] Consider the following KFAE expression.

$$\{\ \texttt{vcc x; x}\ \}(\lambda\texttt{y.y})$$

The following reduction steps show the evaluation process of the given expression. **Complete** the **remaining reduction steps** by filling out the following boxes until the final evaluation result.

$\rightarrow \langle \quad (\varnothing \vdash \{\ \texttt{vcc x; x}\ \}(\lambda\texttt{y.y})) :: \square \ ||$ ■ $\rangle$

$\rightarrow \langle \quad (\varnothing \vdash \{\ \texttt{vcc x; x}\ \}) :: (\varnothing \vdash \lambda\texttt{y.y}) :: (@) :: \square \ ||$ ■ $\rangle$

$\rightarrow \langle \quad ([\texttt{x} \mapsto \langle \kappa_0 \ || \ \blacksquare \rangle] \vdash \texttt{x}) :: (\varnothing \vdash \lambda\texttt{y.y}) :: (@) :: \square \ ||$ ■ $\rangle$

$\rightarrow \langle \quad (\varnothing \vdash \lambda\texttt{y.y}) :: (@) :: \square \ ||$ [ ] $\rangle$

$\rightarrow \langle \quad (@) :: \square \ ||$ [ ] $\rangle$

$\rightarrow \langle$ [ ] $||$ [ ] $\rangle$

$\rightarrow \langle$ [ ] $||$ [ ] $\rangle$

$\rightarrow \langle$ [ ] $||$ [ ] $\rangle$

$\rightarrow \langle \quad \square \ ||$ [ ] $\rangle$

where $\kappa_0 =$ [ ].

(b) 5 points [★★☆] Write the evaluation results of the following KFAE expressions:

- If the expression e evaluates to a value v, write the value v.
- If the expression e does not terminate, write **"not terminate"**.
- If the expression e throws a run-time error, write **"error"**.

```
vcc f;
val g = λx.λy.(x(y * 2) * 3);
val h = g(f);
val z = h(5) * 7;
z * 11
```

Result: [ ]

```
val f = λx.{
  vcc g;
  val y = { vcc h; g(h) * 2 };
  y(x * 3) * 5
};
{ vcc z; f(7)(z) * 11 }
```

Result: [ ]

3. 10 points [★☆☆] Fill in the blanks in the **type derivation** according to the **typing rules** of TRFAE.

$$\dfrac{\dfrac{\boxed{\texttt{(A)}} \qquad \boxed{\texttt{(B)}} \qquad \boxed{\texttt{(C)}}}{\Gamma_0 \vdash \texttt{if(x < 0) 42 else f(x)} : \texttt{num}} \qquad \dfrac{\texttt{f} \in \text{Domain}(\Gamma_1)}{\Gamma_1 \vdash \texttt{f} : \boxed{\texttt{(D)}}}}{\varnothing \vdash \texttt{def f(x:num) : num = if(x < 0) 42 else f(x); f} : \boxed{\texttt{(D)}}}$$

Note that you can use Γ_0 and Γ_1 in the derivation.

(A) =

(B) =

(C) =

(D) =

Γ_0 =

Γ_1 =

4. 5 points [★★☆] Assume that we revised one of **typing rules** in TFAE from the left to the right:

$$\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{val } x \texttt{ = } e_1\texttt{; } e_2 : \tau_2} \qquad\qquad \dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{val } x \texttt{ = } e_1\texttt{; } e_2 : \tau_2}$$

Is the revised type system still **type sound**? If it is, explain why. If not, give a TFAE expression as a **counterexample** that is well-typed according to the revised type system but causes a run-time type-error.

5. 8 points [★★☆] Write down the **evaluation results** and **types** of the following ATFAE expressions. You only get a score **only if both** evaluation results and types are correct.

   - For evaluation results, follow the instructions below:
     - Write results even if they are not well-typed.
     - Write "**error**" if the result is a run-time error.
     - Write "`A`(v)" if the result is a variant value constructed by the constructor `A` with a value v.
     - Write "**not terminate**" if the expression does not terminate.
     - Otherwise, write the resulting value.
   - For types, follow the instructions below:
     - Write **"no type"** if the expression is not well-typed.
     - Otherwise, write the resulting type.

   (a) 2 points
   ```
   enum A { case B(num); }; B(42)
   ```
   Result: [ ] Type: [ ]

   (b) 2 points
   ```
   enum A { case B(bool); };
   enum A { case C(num); };
   B(true)
   ```
   Result: [ ] Type: [ ]

   (c) 2 points
   ```
   enum A { case B(bool); };
   def f(x : A) : num → num = f(x);
   f(B(true))(42)
   ```
   Result: [ ] Type: [ ]

   (d) 2 points
   ```
   enum A { case B(); case C(A, A); };
   def f(x : A) : num = x match {
     case B() => 0;
     case C(y, z) => 1 + (if(f(y) < f(z)) f(z) else f(y));
   };
   f(C(C(B(), C(B(), B())), C(B(), B())))
   ```
   Result: [ ] Type: [ ]

6. 7 points [★★☆] FAE is an **untyped** version of TFAE, and consider the following FAE expression:

   ```
   val twice = λ(f).λ(x).f(f(x));
   val inc = λ(x).(x + 1);
   val x = twice(inc)(42);
   val y = twice(twice)(inc);
   twice(λx.x)(twice);
   ```

   Fill in the blanks to define its **typed** version with **parametric polymorphism** in PTFAE.

   ```
   val twice = ∀α.λ(f : [ ]).λ(x : [ ]).f(f(x));
   val inc = λ(x : num).(x + 1);
   val x = twice [ ] (inc)(42);
   val y = twice [ ] (twice [ ] )(inc);
   twice [ ] (λ(x : [ ]).x)(twice);
   ```

7. 10 points [★☆☆] The **type inference** algorithm infers types without type annotations by generating **type variables** and solving them using **type constraints** collected during type checking.

$$\begin{array}{lll} \text{Type Variables} & \alpha & \in \mathbb{X}_\alpha \\ \text{Solution} & \psi & \in \Psi = \mathbb{X}_\alpha \xrightarrow{\text{fin}} (\mathbb{T} \uplus \{\bullet\}) \end{array}$$

where $\bullet$ represents a not-yet solved type variable.

The $\texttt{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightharpoonup \Psi$ function takes two types and a solution as input and produces a new solution by unifying the two types according to the given solution. Note that it is a partial function because unification may fail when the two types are not unifiable.

$$\boxed{\texttt{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightharpoonup \Psi}$$

$$\texttt{unify}(\tau_1, \tau_2, \psi) = \begin{cases} \psi & \text{if } \tau_1' = \texttt{num} \wedge \tau_2' = \texttt{num} \\ \psi & \text{if } \tau_1' = \texttt{bool} \wedge \tau_2' = \texttt{bool} \\ \texttt{unify}(\tau_{1,r}, \tau_{2,r}, \texttt{unify}(\tau_{1,p}, \tau_{2,p}, \psi)) & \text{if } \tau_1' = (\tau_{1,p} \to \tau_{1,r}) \wedge \tau_2' = (\tau_{2,p} \to \tau_{2,r}) \\ \psi & \text{if } \tau_1' = \alpha = \tau_2' \\ \psi[\alpha \mapsto \tau_2'] & \text{if } \tau_1' = \alpha \wedge \neg\texttt{occur}(\alpha, \tau_2') \\ \psi[\alpha \mapsto \tau_1'] & \text{if } \tau_2' = \alpha \wedge \neg\texttt{occur}(\alpha, \tau_1') \end{cases}$$

where $\tau_1' = \texttt{resolve}(\tau_1, \psi)$ and $\tau_2' = \texttt{resolve}(\tau_2, \psi)$, and `resolve` and `occur` are defined as follows:

$$\boxed{\texttt{resolve} : (\mathbb{T} \times \Psi) \to \mathbb{T}}$$

$$\texttt{resolve}(\tau, \psi) = \begin{cases} \texttt{resolve}(\tau', \psi) & \text{if } \tau = \alpha \wedge \psi(\alpha) = \tau' \\ \texttt{resolve}(\tau_p, \psi) \to \texttt{resolve}(\tau_r, \psi) & \text{if } \tau = (\tau_p \to \tau_r) \\ \tau & \text{otherwise} \end{cases}$$

$$\boxed{\texttt{occur} : (\mathbb{X}_\alpha \times \mathbb{T}) \to \texttt{bool}}$$

$$\texttt{occur}(\alpha, \tau) = \begin{cases} \texttt{true} & \text{if } \tau = \alpha \\ \texttt{occur}(\alpha, \tau_p) \vee \texttt{occur}(\alpha, \tau_r) & \text{if } \tau = (\tau_p \to \tau_r) \\ \texttt{false} & \text{otherwise} \end{cases}$$

For example, after applying the unification algorithm to unify the two types $\alpha \to \texttt{num}$ and $\texttt{bool} \to \beta$ with the initial solution $\{\alpha \mapsto \bullet, \beta \mapsto \bullet\}$, we get the resulting solution:

$$\texttt{unify}(\alpha \to \texttt{num}, \texttt{bool} \to \beta, \{\alpha \mapsto \bullet, \beta \mapsto \bullet\}) = \{\alpha \mapsto \texttt{bool}, \beta \mapsto \texttt{num}\}$$

Fill in the blanks in the **resulting solutions** of the unification algorithm. If it fails, write X.

$\texttt{unify}(\alpha, \beta \to \texttt{num}, \{\alpha \mapsto \bullet, \beta \mapsto \alpha\}) =$ ☐

$\texttt{unify}(\alpha, \texttt{num}, \{\alpha \mapsto \beta, \beta \mapsto \gamma, \gamma \mapsto \bullet\}) =$ ☐

$\texttt{unify}(\alpha \to \beta, \texttt{num} \to (\texttt{bool} \to \texttt{num}), \{\alpha \mapsto \bullet, \beta \mapsto \bullet\}) =$ ☐

$\texttt{unify}(\alpha \to \texttt{num}, \beta, \{\alpha \mapsto \bullet, \beta \mapsto (\texttt{num} \to \gamma), \gamma \mapsto \alpha\}) =$ ☐

$\texttt{unify}(\alpha, \texttt{num} \to \gamma, \{\alpha \mapsto (\beta \to \beta), \beta \mapsto \bullet, \gamma \mapsto \texttt{bool}\}) =$ ☐

8. 15 points STFAE supports **subtype polymorphism** with the **subtype relation** (<:) between types.

   (a) 5 points [★☆☆] Fill in the blanks to complete the following derivation of the **subtype relation**.

   $$\frac{\qquad\qquad}{\{\ \mathtt{a} : \top \to \mathtt{bool},\ \mathtt{b} : \bot\ \} <: \{\ \mathtt{a} : \mathtt{num} \to \top\ \}}$$

   (b) 5 points [★★☆] Assume that the subtype relation for function types in STFAE is modified from the left to the right, which flips the direction of the subtyping relation for the parameter types.

   $$\frac{\tau_1 :> \tau_1' \qquad \tau_2 <: \tau_2'}{(\tau_1 \to \tau_2) <: (\tau_1' \to \tau_2')} \qquad\qquad \frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{(\tau_1 \to \tau_2) <: (\tau_1' \to \tau_2')}$$

   Is the modified subtype relation still **sound** with respect to the typing rules of STFAE? If it is, explain why. If not, give a **counterexample** expression that is well-typed according to the typing rules of STFAE but causes a run-time type-error due to the modified subtype relation.

   (c) 5 points [★★☆] The current typing rule supports subtyping using the **subsumption rule** as follows:

   $$\frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

   However, since it is **non-algorithmic**, some expressions may have **multiple type derivations**. Discuss whether there is an expression that has **infinitely many** type derivations due to the subsumption rule. If such an expression exists, provide an example and explain why. If not, explain why not.

9. [15 points] The following typed language is defined with **array creation** $\langle\tau\rangle[e,\ldots,e]$, **array access** $e[e]$, **array type** $\langle\tau\rangle[n]$ with length n, and **interval types** `[`l`,` h`]` for integers between l and h.

Expressions	$\mathbb{E} \ni e ::= n \mid e$ `+` $e \mid$ `val` $x[:\tau]^?$ `=` e`;` $e \mid x \mid \lambda x.e \mid e(e) \mid \langle\tau\rangle[e,\ldots,e] \mid e[e]$
Types	$\mathbb{T} \ni \tau ::=$ `num` $\mid$ `[`n`,` n`]` $\mid \tau \to \tau \mid \langle\tau\rangle[n] \mid \top$
Type Environments	$\Gamma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}$

The following is an excerpt of the Scala implementation of the type checker for the above language.

```
enum Expr:
  case Num(number: BigInt)
  case Add(left: Expr, right: Expr)
  case Val(name: String, tyOpt: Option[Type], init: Expr, body: Expr)
  case Id(name: String)
  case Fun(param: String, ty: Type, body: Expr)
  case App(fun: Expr, arg: Expr)
  case Array(ty: Type, elems: List[Expr])
  case Access(array: Expr, index: Expr)
enum Type:
  case NumT
  case IntT(low: BigInt, high: BigInt)
  case ArrowT(paramTy: Type, retTy: Type)
  case ArrayT(elemTy: Type, size: BigInt)
  case TopT
type TypeEnv = Map[String, Type]

// subtype relation
def isSubType(lty: Type, rty: Type): Boolean = (lty, rty) match
  case _ if lty == rty                  => true
  case (_, TopT)                        => true
  case (ArrowT(lp, lr), ArrowT(rp, rr)) => isSubType(rp, lp) && isSubType(lr, rr)
  case (IntT(ll, lh), IntT(rl, rh))     => ll >= rl && lh <= rh
  case (IntT(_, _), NumT)               => true
  case (ArrayT(l, n), ArrayT(r, m))     => isSubType(l, r) && n == m
  case _                                => false

// helper function to enforce subtyping
def mustSubType(lty: Type, rty: Type): Unit =
  if (!isSubType(lty, rty)) error(s"type mismatch: ${lty.str} </: ${rty.str}")

// type checker
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  case Num(n) => IntT(n, n)
  case Add(left, right) => (typeCheck(left, tenv), typeCheck(right, tenv)) match
    case (IntT(ll, lh), IntT(rl, rh))     => IntT(ll + rl, lh + rh)
    case (_: IntT | NumT, _: IntT | NumT) => NumT
    case (l, r)                           => error(s"invalid op: ${l.str} + ${r.str}")
  ... // omitted cases
  case Array(ty, elems) =>
    for { elem <- elems } mustSubType(typeCheck(elem, tenv), ty)
    ArrayT(ty, elems.length)
  case Access(array, index) => (typeCheck(array, tenv), typeCheck(index, tenv)) match
    case (ArrayT(elemTy, size), IntT(il, ih)) =>
      if (il < 0 || ih >= size) error(s"index out of bounds: [$il, $ih]")
      elemTy
    case (at, it) => error(s"invalid array access: ${at.str}[${it.str}]")
```

(a) [6 points] [★☆☆] **Define** the **subtype relation** $<:$ according to the `isSubType` function in the given Scala implementation. Define inference rules in the $\boxed{\tau_1 <: \tau_2}$ form.

(b) [4 points] [★★☆] **Define** the **algorithmic typing rules** for numbers n (i.e., `Num`), addition e_1 `+` e_2 (i.e., `Add`), array creation $\langle\tau\rangle[e, \ldots, e]$ (i.e., `Array`), and array access $e[e]$ (i.e., `Access`) according to the `typeCheck` function in the given Scala implementation. Define inference rules in the $\boxed{\Gamma \vdash e : \tau}$ form.

Hint: In the typing rule for addition, you can use the partial function $\mathsf{add} : \mathbb{T} \times \mathbb{T} \rightharpoonup \mathbb{T}$ defined as:

$$\mathsf{add}(\tau_1, \tau_2) = \begin{cases} \texttt{[}n_l + n'_l\texttt{,}\ n_h + n'_h\texttt{]} & \text{if } \tau_1 = \texttt{[}n_l\texttt{,}\ n_h\texttt{]} \wedge \tau_2 = \texttt{[}n'_l\texttt{,}\ n'_h\texttt{]} \\ \texttt{num} & \text{else if } \tau_1 <: \texttt{num} \wedge \tau_2 <: \texttt{num} \end{cases}$$

(c) [5 points] [★★★] If we extend this language to support **mutation** of array elements with the following typing rule, is the type system still **sound**? If it is, explain why. If not, give a **counterexample** expression that is well-typed according to the typing rules of the extended language but causes a run-time type-error due to mutation. (You can also use sequence of expressions $e_1; e_2$.)

$$\frac{\Gamma \vdash e_a : \langle\tau\rangle[n] \qquad \Gamma \vdash e_i : \texttt{[}n_l\texttt{,}\ n_h\texttt{]} \qquad \Gamma \vdash e : \tau' \qquad \tau' <: \tau \qquad 0 \le n_l \qquad n_h < n}{\Gamma \vdash (e_a[e_i] \texttt{ = } e) : \tau'}$$

10. 5 points [★★★] We will design a typed language that supports **pairs** and **linear types** to ensure that all declared **variables** are **used exactly once**. First, the language syntax is defined as follows:

$$\begin{array}{lll} \text{Expressions} & \mathbb{E} \ni e ::= n \mid b \mid x \mid e \texttt{ + } e \mid (e, e) \mid \texttt{val } x \texttt{ = } e\texttt{; } e \mid \texttt{val } (x, y) \texttt{ = } e\texttt{; } e \\ & \qquad \mid \lambda x\texttt{:}\tau.e \mid e(e) \mid \texttt{if } (e) \ e \ \texttt{else } e \\ \text{Types} & \mathbb{T} \ni \tau ::= \texttt{num} \mid \texttt{bool} \mid \tau \to \tau \mid (\tau, \tau) \end{array}$$

Type Environments $\Gamma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}$ Numbers $n \in \mathbb{Z}$ Booleans $b \in \mathbb{B}$ Identifiers $x \in \mathbb{X}$

The operational semantics in the $\boxed{\sigma \vdash e \Rightarrow v}$ form is defined as follows:

$$\frac{}{\sigma \vdash n \Rightarrow n} \qquad \frac{}{\sigma \vdash b \Rightarrow b} \qquad \frac{x \in \text{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)} \qquad \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash (e_1, e_2) \Rightarrow (v_1, v_2)}$$

$$\frac{\sigma \vdash e \Rightarrow v \qquad \sigma[x \mapsto v] \vdash e' \Rightarrow v'}{\sigma \vdash \texttt{val } x \texttt{ = } e\texttt{; } e' \Rightarrow v'} \qquad \frac{\sigma \vdash e \Rightarrow (v_1, v_2) \qquad \sigma[x \mapsto v_1, y \mapsto v_2] \vdash e' \Rightarrow v'}{\sigma \vdash \texttt{val } (x, y) \texttt{ = } e\texttt{; } e' \Rightarrow v'}$$

$$\frac{}{\sigma \vdash \lambda x\texttt{:}\tau.e \Rightarrow \langle \lambda x.e, \sigma \rangle} \qquad \frac{\sigma \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle \qquad \sigma \vdash e_2 \Rightarrow v \qquad \sigma'[x \mapsto v] \vdash e \Rightarrow v'}{\sigma \vdash e_1(e_2) \Rightarrow v'}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2} \qquad \frac{\sigma \vdash e_0 \Rightarrow \texttt{true} \qquad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{if } (e_0) \ e_1 \ \texttt{else } e_2 \Rightarrow v_1} \qquad \frac{\sigma \vdash e_0 \Rightarrow \texttt{false} \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{if } (e_0) \ e_1 \ \texttt{else } e_2 \Rightarrow v_2}$$

where

$$\text{Values} \quad \mathbb{V} \ni v ::= n \mid b \mid (v, v) \mid \langle \lambda x.e, \sigma \rangle \qquad \text{Environments} \quad \sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}$$

Now, let's design the **algorithmic typing rules** to ensure that all declared variables are **used exactly once**. For example, the following expression is well-typed because both `x` and `y` are used exactly once.

$$\texttt{val x = 10; val y = 20; x + y}$$

However, the following expressions are ill-typed because the variable `x` is not used:

$$\texttt{val x = 10; 42}$$

or the variable `y` is used twice:

$$\texttt{val y = 10; y + y}$$

The key idea is to **drop used variables** from the type environment and propagate the remaining type environment after type checking to ensure that all variables are used exactly once. Thus, we use the following form for typing judgments:

$$\boxed{\Gamma \vdash e : \tau \dashv \Gamma'}$$

where Γ is the input type environment, τ is the type of expression e, and Γ' is the output type environment after type checking e.

For example, the following typing judgment states that under the type environment $\{\texttt{x} : \texttt{num}, \texttt{y} : \texttt{num}, \texttt{z} : \texttt{num}\}$, the expression `x + y` has the type `num`, and after type checking, the remaining type environment is $\{\texttt{z} : \texttt{num}\}$ after dropping $\texttt{x} : \texttt{num}$ and $\texttt{y} : \texttt{num}$ because both variables are used during type checking.

$$\{\texttt{x} : \texttt{num}, \texttt{y} : \texttt{num}, \texttt{z} : \texttt{num}\} \vdash \texttt{x + y} : \texttt{num} \dashv \{\texttt{z} : \texttt{num}\}$$

However, the following typing judgment is invalid because `x` is used twice:

$$\{\texttt{x} : \texttt{num}\} \vdash \texttt{x + x} : \texttt{num} \dashv \text{(invalid to drop x twice)}$$

Please define the algorithmic typing rules for the following expressions:

- Identifier Lookup x
- Pair Creation (e, e)
- Variable Declaration `val` x `=` e; e
- Pair Pattern Matching `val` (x, y) `=` e; e
- Lambda Function λx`:`$\tau.e$
- Conditional Expression `if` (e) e `else` e

Note that this language does not allow **variable shadowing**, so you can assume that all variable names are unique. Use $\Gamma \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ to denote the type environment after removing the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ from Γ.

This is the last page.
I hope that your tests went well!

Appendix

TFAE – Typed Functions and Arithmetic Conditional Expressions

Expressions $\mathbb{E} \ni e ::= n \mid b \mid x \mid e \texttt{ + } e \mid e \texttt{ * } e \mid e \texttt{ < } e \mid \texttt{val } x \texttt{ = } e\texttt{; } e \mid \lambda([x\texttt{:}\tau]^*).e \mid e(e^*) \mid \texttt{if } (e)\ e\ \texttt{else } e$
Types $\mathbb{T} \ni \tau ::= \texttt{num} \mid \texttt{bool} \mid (\tau^*) \to \tau$ Booleans $b \in \mathbb{B}$ Numbers $n \in \mathbb{Z}$ Identifiers $x \in \mathbb{X}$

Operational Semantics $\boxed{\sigma \vdash e \Rightarrow v}$

$$\frac{}{\sigma \vdash n \Rightarrow n} \quad \frac{}{\sigma \vdash b \Rightarrow b} \quad \frac{x \in \text{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2} \quad \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ * } e_2 \Rightarrow n_1 \times n_2}$$

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ < } e_2 \Rightarrow n_1 < n_2} \quad \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{val } x \texttt{ = } e_1\texttt{; } e_2 \Rightarrow v_2}$$

$$\frac{}{\sigma \vdash \lambda(x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n).e \Rightarrow \langle \lambda(x_1, \ldots, x_n).e, \sigma \rangle}$$

$$\frac{\begin{array}{c}\sigma \vdash e_0 \Rightarrow \langle \lambda(x_1, \ldots, x_n).e, \sigma' \rangle \\ \sigma \vdash e_1 \Rightarrow v_1 \qquad \ldots \qquad \sigma \vdash e_n \Rightarrow v_n \qquad \sigma'[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \vdash e \Rightarrow v\end{array}}{\sigma \vdash e_0(e_1, \ldots, e_n) \Rightarrow v}$$

$$\frac{\sigma \vdash e_0 \Rightarrow \texttt{true} \qquad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{if } (e_0)\ e_1\ \texttt{else } e_2 \Rightarrow v_1} \quad \frac{\sigma \vdash e_0 \Rightarrow \texttt{false} \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{if } (e_0)\ e_1\ \texttt{else } e_2 \Rightarrow v_2}$$

Values $\mathbb{V} \ni v ::= n \mid b \mid \langle \lambda(x_1, \ldots, x_n).e, \sigma \rangle$ Environments $\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}$

Typing Rules $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash n : \texttt{num}} \quad \frac{}{\Gamma \vdash b : \texttt{bool}} \quad \frac{x \in \text{Domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{num}} \quad \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 \texttt{ * } e_2 : \texttt{num}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash e_1 \texttt{ < } e_2 : \texttt{bool}} \quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{val } x \texttt{ = } e_1\texttt{; } e_2 : \tau_2}$$

$$\frac{\Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau}{\Gamma \vdash \lambda(x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n).e : (\tau_1, \ldots, \tau_n) \to \tau}$$

$$\frac{\Gamma \vdash e_0 : (\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_0(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma \vdash e_0 : \texttt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if } (e_0)\ e_1\ \texttt{else } e_2 : \tau}$$

Type Environments $\Gamma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}$

TRFAE – TFAE with Recursion

$$\text{Expressions} \quad \mathbb{E} \ni e ::= \ldots \mid \texttt{def}\ x([x\texttt{:}\tau]^*)\texttt{:}\tau\ \texttt{=}\ e;\ e$$

Operational Semantics $\boxed{\sigma \vdash e \Rightarrow v}$

$$\ldots \qquad \frac{\sigma' = \sigma[x_0 \mapsto \langle \lambda(x_1, \ldots, x_n).e, \sigma' \rangle] \qquad \sigma' \vdash e' \Rightarrow v'}{\sigma \vdash \texttt{def}\ x_0(x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n)\texttt{:}\tau\ \texttt{=}\ e;\ e' \Rightarrow v'}$$

Typing Rules $\boxed{\Gamma \vdash e : \tau}$

$$\ldots \qquad \frac{\Gamma[x_0 : (\tau_1, \ldots, \tau_n) \to \tau, x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau \qquad \Gamma[x_0 : (\tau_1, \ldots, \tau_n) \to \tau] \vdash e' : \tau'}{\Gamma \vdash \texttt{def}\ x_0(x_1\texttt{:}\tau_1, \ldots, x_n\texttt{:}\tau_n)\texttt{:}\tau\ \texttt{=}\ e;\ e' : \tau'}$$

ATFAE – TRFAE with Algebraic Data Types

$$\text{Expressions} \quad \mathbb{E} \ni e ::= \ldots \mid \texttt{enum}\ t\ \{\ [\texttt{case}\ x(\tau^*)]^*\ \};\ e \mid e\ \texttt{match}\ \{\ [\texttt{case}\ x(x^*)\ \texttt{=>}\ e]^*\ \}$$
$$\text{Types} \quad \mathbb{T} \ni \tau ::= \ldots \mid t \qquad\qquad \text{Type Names} \quad t \in \mathbb{X}_t$$

Operational Semantics $\boxed{\sigma \vdash e \Rightarrow v}$

$$\ldots \qquad \frac{\sigma \vdash e_0 \Rightarrow \langle x \rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \ldots \qquad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash e_0(e_1, \ldots, e_n) \Rightarrow x(v_1, \ldots, v_n)}$$

$$\frac{\sigma[x_1 \mapsto \langle x_1 \rangle, \ldots, x_n \mapsto \langle x_n \rangle] \vdash e \Rightarrow v}{\sigma \vdash \texttt{enum}\ t\ \{\ \texttt{case}\ x_1(\tau_{1,1}, \ldots, \tau_{1,m_1});\ \ldots;\ \texttt{case}\ x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})\ \};\ e \Rightarrow v}$$

$$\frac{\sigma \vdash e \Rightarrow x_i(v_1, \ldots, v_{m_i}) \qquad \forall j < i.\ x_j \neq x_i \qquad \sigma[x_{i,1} \mapsto v_1, \ldots, x_{i,m_i} \mapsto v_{m_i}] \vdash e_i \Rightarrow v}{\sigma \vdash e\ \texttt{match}\ \{\ \texttt{case}\ x_1(x_{1,1}, \ldots, x_{1,m_1})\ \texttt{=>}\ e_1;\ \ldots;\ \texttt{case}\ x_n(x_{n,1}, \ldots, x_{n,m_n})\ \texttt{=>}\ e_n\ \} \Rightarrow v}$$

$$\text{Values} \quad \mathbb{V} \ni v ::= \ldots \mid \langle x \rangle \mid x(v^*)$$

Typing Rules $\boxed{\Gamma \vdash e : \tau}$

$$\ldots \qquad \frac{\begin{array}{c}\Gamma' = \Gamma[t = x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})] \qquad t \notin \text{Domain}(\Gamma) \\ \Gamma' \vdash \tau_{1,1} \quad \ldots \quad \Gamma' \vdash \tau_{n,m_n} \qquad \Gamma'[x_1 : (\tau_{1,1}, \ldots, \tau_{1,m_1}) \to t, \ldots, x_n : (\tau_{n,1}, \ldots, \tau_{n,m_n}) \to t] \vdash e : \tau \qquad \Gamma \vdash \tau\end{array}}{\Gamma \vdash \texttt{enum}\ t\ \{\ \texttt{case}\ x_1(\tau_{1,1}, \ldots, \tau_{1,m_1});\ \ldots;\ \texttt{case}\ x_n(\tau_{n,1}, \ldots, \tau_{n,m_n})\ \};\ e : \tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \qquad \Gamma(t) = x_1(\tau_{1,1}, \ldots, \tau_{1,m_1}) + \ldots + x_n(\tau_{n,1}, \ldots, \tau_{n,m_n}) \\ \Gamma[x_{1,1} : \tau_{1,1}, \ldots, x_{1,m_1} : \tau_{1,m_1}] \vdash e_1 : \tau \qquad \ldots \qquad \Gamma[x_{n,1} : \tau_{n,1}, \ldots, x_{n,m_n} : \tau_{n,m_n}] \vdash e_n : \tau\end{array}}{\Gamma \vdash e\ \texttt{match}\ \{\ \texttt{case}\ x_1(x_{1,1}, \ldots, x_{1,m_1})\ \texttt{=>}\ e_1;\ \ldots;\ \texttt{case}\ x_n(x_{n,1}, \ldots, x_{n,m_n})\ \texttt{=>}\ e_n\ \} : \tau}$$

$$\text{Type Environments} \quad \Gamma \in (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}) \times (\mathbb{X}_t \xrightarrow{\text{fin}} (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}^*))$$

Well-formedness of Types $\boxed{\Gamma \vdash \tau}$

$$\frac{}{\Gamma \vdash \texttt{num}} \qquad \frac{}{\Gamma \vdash \texttt{bool}}$$

$$\frac{\Gamma \vdash \tau_1 \quad \ldots \quad \Gamma \vdash \tau_n \qquad \Gamma \vdash \tau}{\Gamma \vdash (\tau_1, \ldots, \tau_n) \to \tau} \qquad \frac{\Gamma(t) = x_1(\ldots) + \ldots + x_n(\ldots)}{\Gamma \vdash t}$$

PTFAE – TFAE with Parametric Polymorphism

Expressions $\mathbb{E} \ni e ::= \ldots \mid \forall\alpha.e \mid e[\tau]$ Types $\mathbb{T} \ni \tau ::= \ldots \mid \forall\alpha.\tau \mid \alpha$ Type Variables $\alpha \in \mathbb{X}_\alpha$

Note that this language restricts the number of function parameters to one for simplicity.

Operational Semantics $\boxed{\sigma \vdash e \Rightarrow v}$

$$\ldots \qquad \frac{}{\sigma \vdash \forall\alpha.e \Rightarrow \langle \forall\alpha.e, \sigma\rangle} \qquad \frac{\sigma \vdash e \Rightarrow \langle \forall\alpha.e', \sigma'\rangle \qquad \sigma' \vdash e' \Rightarrow v'}{\sigma \vdash e[\tau] : v'}$$

Values $\mathbb{V} \ni v ::= \ldots \mid \langle \forall\alpha.e, \sigma\rangle$

Typing Rules $\boxed{\Gamma \vdash e : \tau}$

$$\ldots \qquad \frac{\alpha \notin \text{Domain}(\Gamma) \qquad \Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash \forall\alpha.e : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash e : \forall\alpha.\tau'}{\Gamma \vdash e[\tau] : \tau'[\alpha \leftarrow \tau]}$$

Type Environments $\Gamma \in (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}) \times (\mathbb{X}_t \xrightarrow{\text{fin}} (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}^*)) \times \mathcal{P}(\mathbb{X}_\alpha)$

Well-formedness of Types $\boxed{\Gamma \vdash \tau}$

$$\frac{}{\Gamma \vdash \texttt{num}} \qquad \frac{}{\Gamma \vdash \texttt{bool}} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \tau'}{\Gamma \vdash \tau \to \tau'} \qquad \frac{\Gamma[\alpha] \vdash \tau}{\Gamma \vdash \forall\alpha.\tau} \qquad \frac{\alpha \in \text{Domain}(\Gamma)}{\Gamma \vdash \alpha}$$

STFAE – TFAE with Records and Subtype Polymorphism

Expressions $\mathbb{E} \ni e ::= \ldots \mid \{[x = e]^*\} \mid e.x \mid \texttt{exit}$ Types $\mathbb{T} \ni \tau ::= \ldots \mid \{[x : \tau]^*\} \mid \bot \mid \top$

Note that this language restricts the number of function parameters to one for simplicity.

Operational Semantics $\boxed{\sigma \vdash e \Rightarrow v}$

$$\ldots \qquad \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \ldots \qquad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash \{x_1 = e_1, \ldots, x_n = e_n\} \Rightarrow \{x_1 = v_1, \ldots, x_n = v_n\}} \qquad \frac{\sigma \vdash e \Rightarrow \{x_1 = v_1, \ldots, x_n = v_n\} \qquad 1 \le i \le n}{\sigma \vdash e.x_i \Rightarrow v_i}$$

Values $\mathbb{V} \ni v ::= \ldots \mid \{[x = v]^*\}$

Typing Rules $\boxed{\Gamma \vdash e : \tau}$

$$\ldots \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{x_1 = e_1, \ldots, x_n = e_n\} : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

$$\frac{\Gamma \vdash e : \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \qquad 1 \le i \le n}{\Gamma \vdash e.x_i : \tau_i} \qquad \frac{}{\Gamma \vdash \texttt{exit} : \bot}$$

Subtype Relation $\boxed{\tau <: \tau}$

$$\frac{}{\bot <: \tau} \qquad \frac{}{\tau <: \top} \qquad \frac{}{\tau <: \tau} \qquad \frac{\tau <: \tau' \qquad \tau' <: \tau''}{\tau <: \tau''} \qquad \frac{\tau_1 :> \tau_1' \qquad \tau_2 <: \tau_2'}{(\tau_1 \to \tau_2) <: (\tau_1' \to \tau_2')}$$

$$\frac{}{\{x_1 : \tau_1, \ldots, x_n : \tau_n, x : \tau\} <: \{x_1 : \tau_1, \ldots, x_n : \tau_n\}} \qquad \frac{\tau_1 <: \tau_1' \qquad \ldots \qquad \tau_n <: \tau_n'}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} <: \{x_1 : \tau_1', \ldots, x_n : \tau_n'\}}$$

$$\frac{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \text{ is a permutation of } \{x_1' : \tau_1', \ldots, x_n' : \tau_n'\}}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} <: \{x_1' : \tau_1', \ldots, x_n' : \tau_n'\}}$$