ESTR3106 Cheatsheet

0. Scope 与做题顺序

不考推理：denotational / axiomatic semantics；natural deduction rules。Toy language 的 syntax / inference rules 会给，不用背全规则。
做题先后：syntax valid? -> value/final? -> 有 small-step rule? -> 无规则且非 value = stuck；一直可 step 不停 = diverge；同一配置多于一个 next step = nondeterministic。

高频题型	一句话模板
type safe	well-typed programs do not get stuck = progress + preservation
not type safe	找 `well-typed` 但 `stuck` 的反例
typing derivation	按规则自底向上堆树；lambda 无标注时可选能使推导成立的参数类型
small-step	一次只走一步；注意 evaluation order 和 side effects
Prolog	facts + rules + query；unification + DFS backtracking

1. Operational Semantics

概念	速记
small-step	`(e,s) -> (e',s')`，描述一步执行
big-step	`(e,s) => (v,s')`，直接到最终值；普通 big-step 难区分 stuck/diverge
value	已完成的表达式；Simpl 中 `skip` 也可视为 value/final
stuck	非 value 且没有规则可用，如 type error / missing case
diverge	一直有下一步但永远不终止，如 `while true do skip`、`Ω`
deterministic	每个配置至多一个 next step

求值顺序有副作用会影响 store：`(x:=3; !x) + (x:=4; !x)` 值都是 `7`；left-to-right 最终 `x=4`，right-to-left 最终 `x=3`。
Short-circuit：`true or e -> true`、`false and e -> false`，右边不执行，右边副作用不发生。
语法糖/desugaring 不是运行一步：`e1 - e2 ≡ e1 + (-1 * e2)`；`not e ≡ if e then false else true`；`e1 or e2 ≡ if e1 then true else e2`；`e1 and e2 ≡ if e1 then e2 else false`。

2. Types 与 Type Safety

记号	意义
`Γ ⊢ e : τ`	在上下文 `Γ` 下，`e` 有类型 `τ`
`Γ, x:τ1`	在检查函数体时把参数 `x` 加入 context；不是多态
`τ1 -> τ2`	函数类型：参数 `τ1`，返回 `τ2`
`unit`	只关心副作用的类型；`skip : unit`
type checking	给定/标注类型后检查
type inference	自动找 `τ` 使 `Γ ⊢ e : τ`

Type safe = Progress + Preservation：

性质	标准句
Progress	If `⊢ e : τ`, then `e` is a value or exists `e'` such that `e -> e'`.
Preservation	If `⊢ e : τ` and `e -> e'`, then `⊢ e' : τ`.
结论	well-typed programs do not get stuck

证明模板：Progress 用 induction on typing derivation `⊢ e:τ`；每个 typing rule 分 case，用 IH 证明子表达式 value 或可 step。Preservation 用 induction on `e -> e'` 或 typing derivation；每个 evaluation rule 分 case，证明 step 后类型不变。
设计类型系统：太强也可 type safe 但无用，例如只给整数 typing rule，则 preservation vacuous、progress trivial，但拒绝 `1*2` / `λx.x` 等安全程序。
Store 类型安全：store = location -> value；location 是可读写位置；aliasing = 多个变量/引用指向同一 location。带 store 时需保证 typing context 与 store 一致：`Σ ⊢ s` / store typing。

3. Induction / Substitution / Lambda

IH = induction hypothesis。结构归纳按语法子结构；推导树归纳按最后一条 rule 分 case。常见句：`Proof by induction on the derivation of Γ ⊢ e : τ.`

概念	速记
binder/scope	`λx.e` / `[x]e` 绑定 `x`；scope 尽量向右延伸
free variable	没被当前 binder 绑定的变量出现
α-equivalence	只改 bound variable 名字：`λx.x ≡α λy.y`
β-reduction	`(λx.e1) e2 -> e1[e2/x]`
capture-avoiding substitution	避免把自由变量替换成被内层 binder 捕获；必要时先 alpha-renaming

CBV/CBN/lazy：CBV 先把参数求成 value；CBN 直接代入，用到再算；lazy/call-by-need 用到才算且缓存一次。`(λx.1) Ω`：CBN/lazy 得 `1`，CBV 先算 `Ω` 而 diverge。
`Ω = (λx.x x)(λx.x x)`，一步后回到自身，无 normal form。
Y combinator：`Y = λf.(λx.f(x x))(λx.f(x x))`，性质 `Y f ≈ f(Y f)`。CBV 下普通 Y 会先展开 `A A`，还没进入函数体 base case 就 diverge。
Z combinator：`Z = λf.(λx.f(λy.x x y))(λx.f(λy.x x y))`。额外 `λy` 把 self-application 延迟，适合 CBV。Typed Py / simply-typed 系统通常不能给 `x(x)` 类型，因为要求 `α = α -> β`。
Church numeral：`0=λf.λx.x`，`1=λf.λx.f x`，`2=λf.λx.f(f x)`。加法：`λn.λm.λf.λx. n f (m f x)`，先做 `m` 次，再做 `n` 次。
Church-Rosser / confluence：不同 reduction 路径可汇合；若 normal form 存在，则唯一到 alpha-renaming。
Curry-Howard：命题即类型，证明即程序；构造出某类型的程序就像证明对应命题。

4. Polymorphism / Subtyping / Data

概念	速记
parametric polymorphism	同一代码对任意类型统一工作：`'a -> 'a`
ad-hoc polymorphism	同名按类型/arity 选实现，如 overload、`tree/1` 与 `tree/2`
subtype polymorphism	subtype 可用在需要 supertype 的地方
product type	`τ1 * τ2`，pair/tuple
sum / ADT	一个值属于多个 variant 之一，可携带不同数据
nominal typing	看显式类型名/类名
structural typing	看字段/方法结构是否匹配

Subtyping 核心：`τ <: τ'` 表示 `τ` 的值可安全当作 `τ'` 用。Record：字段更多的 record `<:` 字段更少的 record。Reference subtyping 通常 invariant，否则读写会不安全。
Function subtyping 正确规则：参数反变，返回协变。

```
τ1' <: τ1       τ2 <: τ2'
--------------------------------
τ1 -> τ2  <:  τ1' -> τ2'
```

记忆：需要 `Dog -> int` 时可给 `Animal -> int`，因为能处理所有 Animal 的函数也能处理 Dog。错误的参数协变会 unsound。
Practice 4 反例：若 `zero <: int` 且错误地参数协变，则 `(λf. f 1) (λx. x)` 可被推出 type `zero`，但实际 evaluates to `1`。

5. Practice Patterns

Py / Typed Py:

表达式	动态行为	静态类型
`(λx.1+x)("1")`	`1+"1"` stuck	not well-typed
`(λx.2)("1")`	`2`	well-typed, type `int`；可令 `x:string`
`(λx.x(x)) (λx.x(x))`	diverge	not well-typed: self-application
`(λx.x)((λx.x)(1))`	`1`	well-typed, `int`
`((λx.x)(λx.x))(1)`	`1`	well-typed；不是同一个 `x` 自我应用

Ap pair language：unityped `all` 不安全，反例 `1 + (2,3)` well-typed but stuck。安全设计：`int`；pair type `τ1*τ2`；`+,-,*` 两边同型 `τ`，结果 `τ`。
Week 13 process language：composition `|` right-associative；`[x]e` binds `x`，scope 尽量向右；`<e>` send，`[x]e` receive；`z :: zs` 是非空 list，head `z:int`，tail `zs:int list`。`(skip|stop,[])` 不是 always stuck，因为 nondeterminism 可到 `stop` 或 `skip`。

6. Prolog

语法/机制	速记
fact	`parent(alice,bob).`
rule	`H :- B.` 读作 H if B；`,` 是 and
query	`?- p(X).` 大写开头是变量
identity	predicate = `name/arity`，如 `tree/1`
execution	从左到右、从上到下、DFS + backtracking
unification	`X = 2+3` 绑定结构，不计算
arithmetic	`X is 2+3` 先算右侧，得 `X=5`
failure	`integer(hello).` fail，通常不是 type error
cut	`!` 承诺当前分支，阻止回溯到后续规则

Binary tree / max:

```
tree(X) :- integer(X).
tree((L,R)) :- tree(L), tree(R).

max_tree(X,X) :- integer(X).
max_tree((L,R),M) :-
    max_tree(L,ML), max_tree(R,MR), max2(ML,MR,M).

max2(A,B,B) :- A < B.
max2(A,B,A) :- B < A.
max2(A,A,A).
```

注意 `(L,R)` 是 compound term，不是 list `[L,R]`，也不是给 `tree` 两个参数。

7. Language Comparison

Language	考试速记
OCaml	static + type inference + type safe；compiled successfully => 通常无 static/runtime typing errors；`=` equality，`:=` 写 ref，`!` 读 ref；not pure，有 mutation/I/O
Python	dynamic typing；可编译；运行到坏表达式才 TypeError；objects + first-class functions
C++	static；compiled successfully => 无普通 static type error；pointers/casts/UB 使其不 fully type safe；有 objects/function pointers
Java	static；mostly type safe；casts 可能 runtime `ClassCastException`
Prolog	logic programming；facts/rules/query；unification/backtracking；无普通静态类型检查，很多 mismatch 是 fail

常见判断：statically typed = C++ / OCaml / Java；possible compiler != statically typed；pure functional = none；has objects = C++ / Python / OCaml / Java；pattern matching = OCaml / Prolog。

8. ESTR Practice Exams 1-4 极简题答

Exam 1: Booli + OCaml

Booli syntax：`stmt ::= stmt;stmt | if(bexp)stmt | x:=bexp | skip`；`bexp ::= true|false|!x|bexp or bexp`。

Q	题目要点	答案
Q1a syntax	判断 5 个 string 是否是 Booli stmt	`x:=true and..` no；`skip;skip` yes；`true or true or true` no，只是 `bexp`；`if(true)x:=true;false` no；`x:=true; x:=x or x` no，需 `!x`
Q1b(i)	semantics deterministic?	nondeterministic；`or` 可先左/右 step；例 `x:=!x or !y` 可到 `x:=false or !y` 或 `x:=!x or true`

Q	题目要点	答案
Q1b(ii)	证明 `if(!x) x:=false` well-typed under `x:bool ref`	`x:bool ref ⊢ !x:bool` ; `false:bool` ; `x:=false:unit` ; 用 TIf 得整体 `unit`
Q1b(iii)	Booli- type safe?	yes。证明 progress + preservation； `skip` final；seq/if/or 用 IH；无 assignment 所以 store 基本不变
Q2	OCaml library compiled 后会有什么 typing error?	通常 no static typing errors；OCaml type safe，所以也不期待 runtime typing errors；仍可能有 non-typing exception / logic bug / nontermination

Seq progress 模板：若 `stmt1=skip` 用 Seq-Skip；否则由 IH 得 `stmt1->stmt1'` ，再推出 `stmt1;stmt2 -> stmt1';stmt2` 。Seq preservation：inversion 得两边 `unit` ，IH 保 `stmt1' : unit` ，再用 TSeq。

Exam 2: Py / Typed Py + Pointer + Church

Py 值：string/int/lambda；CBV。Typed Py：简单静态类型，不支持递归类型。

Q	表达式/任务	答案
Q1a E1	`(λx.1+x)("1")`	steps to `1+"1"` then stuck
Q1a E2	`(λx.2)("1")`	evaluates to `2`
Q1a E3	`(λx.x(x))(λx.x(x))`	diverges
Q1a E4	`(λx.x)((λx.x)(1))`	evaluates to `1`
Q1b	static typing	E1 no；E2 yes `int` ，可令 `x:string` ；E3 no， `x(x)` requires `α=α->β` ；E4 yes `int`
Q1c	OCaml AST	`String of string | Int of int | Var of string | Plus of ast*ast | Lambda of string*ast | App of ast*ast`
Q1d	Z combinator in Typed Py?	No；含 `x(x)` ，simply-typed system 无法赋有限类型；Python 能跑是 dynamic typing
Q2	C++ pointer	`int* x=&y; z=*x` prints `2` ； `*x=1; cout<<y` prints `1` ，aliasing
Q3	`mystery=λn.λm.λf.λx.n f (m f x)`	`mystery 1 2 -> λf.λx.f(f(f x)) = 3` ；功能是 Church addition

Z/Python 考试句：Python checks `x(x)` only at runtime; Typed Py must type-check the expression before running and rejects self-application.

Exam 3: Ap Type Safety + Prolog

Ap： `e ::= z | e1 op e2` ， `op=+|-|*|,` ； `,` builds pairs，运算对同形 pair elementwise。

Q	题目要点	答案
Q1a	Charlie unityped `all` 是否 type safe	No；反例 `1 + (2,3)` well-typed as `all` but stuck
Q1b	设计安全 type system	`τ ::= int | τ1*τ2` ； `z:int` ； `,` 得 product； `+,-,*` 要求两边同一 `τ` ，结果 `τ`
Q1c	证明 type safe	progress + preservation by induction on typing derivation / step；pair case 用 IH；op case 两边同型，value 时有对应计算规则
Q2a	Prolog `tree/1`	`tree(X):-integer(X). tree((L,R)):-tree(L),tree(R).`
Q2b	Prolog max	leaf max 自己；pair 递归求 `ML/MR` 再 `max2`
Q2c	`tree/1` 与 `tree/2` polymorphism	ad-hoc polymorphism / overloading；同名不同 arity

`max2` :

```
max2(A,B,B):- A < B.  max2(A,B,A):- B < A.  max2(A,A,A).
```

Exam 4: Mul + Subtyping + Language Compare

Mul： `e ::= z | x | e1*e2 | λx.e | e1 e2` ，values `z | λx.e` 。

Q	题目要点	答案
Q1a	grammar ambiguous?	yes；如 `1*2*3` 可左/右结合； `x y z` 也可多种 parse
Q1b	given `e -> e'` 是 small/big?	small-step，因为只走一步
Q1c(i)	最少 typable 且 preservation	只给 `(Int) ⊢z:int` ；不 typing var/lambda/app/mul
Q1c(ii)	type safe?	yes；well-typed 只有 integers，已经 value；progress 成立，preservation vacuous
Q1c(iii)	实用吗	no；拒绝 `1*2` 、 `λx.x` 等安全程序
Q1d(i)	给定函数 subtyping 参数方向	covariant in parameter；但正确规则应 contravariant
Q1d(ii)	找 type `zero` 但 evaluates to `1`	`(λf. f 1) (λx. x)`
Q2a	语言性质	static: C++/OCaml；can compile: all three；pure functional: none；objects: C++/Python/OCaml；function pointers: C++
Q2b	哪段能 compile/run	only Python: `x=1; x="hello"` ；C++ `int x; x="hello"` no；OCaml `x=int ref` 与 string 比较 no

Subtyping 反例推导： `λx.x : zero->zero` ：因错误 covariance 且 `zero <: int` ，得 `zero->zero <: int->zero` ，所以 `λx.x : int->zero` ；于是 `(λf.f 1)(λx.x) : zero` ，但运行到 `1` 。

9. Progress + Preservation 答题模板

In this exercise, you will consider the following small-step semantics and type system for the RPN language:
Semantics
$(\text{PLUS-1})\ \frac{e_1 \to e_1'}{e_1 e_2 + \to e_1' e_2 +}$
$(\text{PLUS-2})\ \frac{e_2 \to e_2'}{e_1 e_2 + \to e_1 e_2' +}$
$(\text{PLUS-INT})\ z_1 z_2 + \to z \text{ if } z = z_1 + z_2$
$(\text{PLUS-FLOAT})\ f_1 f_2 + \to f \text{ if } f = f_1 + f_2$
$(\text{TIMES-1})\ \frac{e_1 \to e_1'}{e_1 e_2 * \to e_1' e_2 *}$
$(\text{TIMES-2})\ \frac{e_2 \to e_2'}{e_1 e_2 * \to e_1 e_2' *}$
$(\text{TIMES-INT})\ z_1 z_2 * \to z \text{ if } z = z_1 \times z_2$
$(\text{TIMES-FLOAT}) f_1 f_2 * \to f \text{ if } f = f_1 \times f_2$
Types
$\tau ::= \text{int} \mid \text{float} \mid \text{string}$
Typing Rules
$(\text{T-INT})\ \vdash z : \text{int} \qquad (\text{T-FLOAT})\ \vdash f : \text{float}$
$(\text{T-PLUS})\ \frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1 e_2 + : \tau} \qquad (\text{T-TIMES})\ \frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1 e_2 * : \tau}$
Does the type system above make RPN type-safe? If so, prove it. If not, give a counterexample to type safety.
Solution: Yes, the type system above make RPN type-safe.
Proof: to prove type-safety, just prove progress and preservation.
① The RPN language has the preservation property.
Proof. By induction on the derivation of $\vdash e : \tau$:

- Case (T-INT), Case (T-FLOAT): e is an integer z or float f. It is a value and cannot step, so the property vacuously holds.
- Case (T-PLUS): $e = e_1 e_2 +$ for some e_1, e_2, τ with $\vdash e_1 : \tau$ and $\vdash e_2 : \tau$. By case analysis on $e \to e'$:
  - Case (PLUS-1): $e' = e_1' e_2 +$ and $e_1 \to e_1'$. By IH, $\vdash e_1' : \tau$. By (T-PLUS), $\vdash e_1' e_2 + : \tau$.
  - Case (PLUS-2): $e' = e_1 e_2' +$ and $e_2 \to e_2'$. By IH, $\vdash e_2' : \tau$. By (T-PLUS), $\vdash e_1 e_2' + : \tau$.
  - Case (PLUS-INT): $e_1 = z_1, e_2 = z_2$, and $e' = z$. We already know $\tau = \text{int}$. By (T-INT), $\vdash z : \text{int}$.
  - Case (PLUS-FLOAT): $e_1 = f_1, e_2 = f_2$, and $e' = f$. We already know $\tau = \text{float}$. By (T-FLOAT), $\vdash f : \text{float}$.
- Case (T-TIMES): The proof is identical to Case (T-PLUS), replacing `+` with `*` and invoking (TIMES-1), (TIMES-2), (TIMES-INT), and (TIMES-FLOAT).

② The RPN language has the progress property.
Proof. By induction on the derivation of $\vdash e : \tau$.

- Case (T-INT), Case (T-FLOAT): e is an integer z or a float f. It follows that e is a value.
- Case (T-PLUS): $e = e_1 e_2 +$ for some e_1, e_2, τ with $\vdash e_1 : \tau$ and $\vdash e_2 : \tau$.
  - Case e_1 is not a value: By IH, $e_1 \to e_1'$. By (PLUS-1), $e \to e_1' e_2 +$.
  - Case e_1 is a value, but e_2 is not a value: By IH, $e_2 \to e_2'$. By (PLUS-2), $e \to e_1 e_2' +$.
  - Case e_1 and e_2 are both values: Because there is no `(T-STRING)` introduction rule, the type τ must be either `int` or `float` .
    - If $\tau = \text{int}$, e_1 and e_2 must be integers z_1, z_2 derived by (T-INT). By (PLUS-INT), $e \to z$.
    - If $\tau = \text{float}$, e_1 and e_2 must be floats f_1, f_2 derived by (T-FLOAT). By (PLUS-FLOAT), $e \to f$.
- Case (T-TIMES): The proof is identical to Case (T-PLUS), replacing `+` with `*` and applying the corresponding `TIMES` semantic rules.

Since we have proven progress and preservation, we can conclude that the language is type-safe.

10. 手搓 Type System 模板

This question is about the toy language AP, whose syntax and semantics are shown below. The metavariable z ranges over the set of integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. All operators are left-associative.
$e \in \langle expr \rangle ::= z \mid e_1 \langle op \rangle e_2 \qquad \langle op \rangle ::= + \mid - \mid * \mid ,$
AP Operator Precedence Highest precedence: $*$ Middle precedence: $+, -$ Lowest precedence: ,
The set of values in AP are given below: $v ::= z \mid v_1, v_2$
AP Small-Step Operational Semantics
$(\text{Op-1})\ \frac{e_1 \to e_1'}{e_1 \langle op \rangle e_2 \to e_1' \langle op \rangle e_2} \qquad (\text{Op-2})\ \frac{e \to e'}{v \langle op \rangle e \to v \langle op \rangle e'}$
$(\text{Op-Pair})\ \ (v_1, v_2) \langle op \rangle (v_3, v_4) \to (v_1 \langle op \rangle v_3), (v_2 \langle op \rangle v_4) \quad \text{for } \langle op \rangle \neq ,$
$(\text{Plus})\ \ z_1 + z_2 \to z_3 \quad \text{if } z_3 = z_1 + z_2$
$(\text{Minus})\ \ z_1 - z_2 \to z_3 \quad \text{if } z_3 = z_1 - z_2$
$(\text{Times})\ \ z_1 * z_2 \to z_3 \quad \text{if } z_3 = z_1 * z_2$
Charlie has proposed adding a type system to the AP language. His proposed type system makes AP a *unityped* language – all expressions in the language have a single type **all**, and contains only the inference rule shown below: $\Gamma \vdash e : \textbf{all}$
(a) (10 marks) Give an example demonstrating that the typed version of AP using Charlie's inference rule is not type safe.
(b) (25 marks) Design a new type system for AP that meets the below requirements: It makes AP type-safe; It is such that any AP expression e that evaluates to a value in zero or more steps of the semantics is well-typed in the type system.
(c) (35 marks) Prove that your new type system makes AP type-safe.

Solution
(a) The type system is not type-safe because it violates the **Progress** property. Under Charlie's rule, the expression `1 + (2, 3)` is well-typed ($\Gamma \vdash 1 + (2, 3) : \textbf{all}$). However, it is neither a value nor can it take a step (it gets "stuck"), since neither `(Plus)` nor `(Op-Pair)` can apply to an integer and a pair operand.
(b) We introduce two types: integers and pairs.
Types: $T ::= \text{Int} \mid T_1 \times T_2$
Typing Rules: $(\text{T-Int})\ \frac{}{\Gamma \vdash z : \text{Int}} \qquad (\text{T-Comma})\ \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1, e_2 : T_1 \times T_2}$
$(\text{T-Op})\ \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \langle op \rangle e_2 : T} \quad \text{for } \langle op \rangle \in \{+, -, *\}$
Note: The `(T-Op)` rule ensures that arithmetic operations only occur between structures of the exact same shape.
(c) To prove type safety, we prove Progress and Preservation.
1. Progress: If $\emptyset \vdash e : T$, then e is a value or $e \to e'$.
By structural induction on the derivation of $\emptyset \vdash e : T$:

- **Case (T-Int):** $e = z$, which is already a value.
- **Case (T-Comma):** $e = e_1, e_2$. By IH, e_1 and e_2 are values or can step. If e_1 steps, e steps via `(Op-1)` . If $e_1 = v_1$ and e_2 steps, e steps via `(Op-2)` . If both are values (v_1, v_2), e is a value.
- **Case (T-Op):** $e = e_1 \langle op \rangle e_2$ with $\langle op \rangle \in \{+, -, *\}$. By IH, e_1, e_2 are values or step. If either steps, e steps via `(Op-1)` or `(Op-2)` . If both are values ($e_1 = v_1, e_2 = v_2$), since they share the same type T, their shapes must match (Canonical Forms):
  - If $T = \text{Int}$, $v_1 = z_1$ and $v_2 = z_2$. It can step via `(Plus)` , `(Minus)` , or `(Times)` .
  - If $T = T_1 \times T_2$, $v_1 = (v_{1a}, v_{1b})$ and $v_2 = (v_{2a}, v_{2b})$. It can step via `(Op-Pair)` .

2. Preservation: If $\emptyset \vdash e : T$ and $e \to e'$, then $\emptyset \vdash e' : T$.
By induction on the derivation of $e \to e'$:

- **Case (Op-1) & (Op-2):** $e_1 \langle op \rangle e_2 \to e_1' \langle op \rangle e_2$ (or vice versa). By IH, the stepped subexpression retains its type. Reapplying the respective typing rule yields the original type T.
- **Case (Plus)/(Minus)/(Times):** $z_1 \langle op \rangle z_2 \to z_3$. The original expression has type Int via `(T-Op)` . The result z_3 has type Int via `(T-Int)` . Preserved.
- **Case (Op-Pair):** $(v_1, v_2) \langle op \rangle (v_3, v_4) \to (v_1 \langle op \rangle v_3), (v_2 \langle op \rangle v_4)$.
  - From `(T-Op)` , we know $\vdash (v_1, v_2) : T$ and $\vdash (v_3, v_4) : T$.
  - By `(T-Comma)` inversion, $T = T_1 \times T_2$, where $\vdash v_1 : T_1, \vdash v_2 : T_2$, $\vdash v_3 : T_1$, and $\vdash v_4 : T_2$.
  - Using `(T-Op)` , we get $\vdash v_1 \langle op \rangle v_3 : T_1$ and $\vdash v_2 \langle op \rangle v_4 : T_2$.
  - Using `(T-Comma)` again, we combine them: $\vdash (v_1 \langle op \rangle v_3), (v_2 \langle op \rangle v_4) : T_1 \times T_2$. The type T is preserved.