

Lecture 15 – Simply-Typed Lambda Calculus

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Motivation: Safe Language Systems
 - Detecting Run-Time Errors
 - Dynamic vs Static Analysis
 - Soundness vs Completeness
- Type System
 - Types
 - Type Errors
 - Type Checking
 - Type Soundness

1. Simply-Typed Lambda Calculus

Abstract Syntax

Dynamic Semantics

Static Semantics

2. Type Soundness

Type Soundness

Progress

Preservation

We can define the **abstract syntax** of **simply-typed lambda calculus** (STLC) as follows:

$$\begin{array}{l} \text{Expressions } e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e \\ \text{Types } \tau ::= \text{num} \mid \tau \rightarrow \tau \end{array}$$

We can define the abstract syntax of STLC in Scala as follows:

```
enum Expr:
  case Num(num: Int)
  case Var(name: String)
  case Add(left: Expr, right: Expr)
  case Abs(param: String, ty: Type, body: Expr)
  case App(fun: Expr, arg: Expr)

enum Type:
  case NumT
  case ArrowT(paramTy: Type, retTy: Type)
```

Expressions $e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e$
 Types $\tau ::= \text{num} \mid \tau \rightarrow \tau$
 Values $v ::= \lambda x:\tau. e \mid n$

$e \rightarrow e$

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n} \text{ADD} \qquad \frac{}{(\lambda x:\tau. e)(v) \rightarrow e[x \mapsto v]} \beta\text{-REDUCE}$$

$$\frac{e_1 \rightarrow e_2}{E[e_1] \rightarrow E[e_2]} \text{CONTEXT}$$

$$E ::= [\cdot] \mid E e \mid v E \mid E + e \mid v + E$$

Expressions	$e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e$
Types	$\tau ::= \mathbf{num} \mid \tau \rightarrow \tau$
Values	$v ::= \lambda x:\tau. e \mid n$
Typing Environments	$\Gamma \in \mathbb{X} \rightarrow \tau$

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

$$\frac{}{\Gamma \vdash n : \mathbf{num}} \text{T-NUM} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \text{T-VAR}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{num} \quad \Gamma \vdash e_2 : \mathbf{num}}{\Gamma \vdash e_1 + e_2 : \mathbf{num}} \text{T-ADD}$$

$$\frac{\Gamma[x:\tau] \vdash e : \tau'}{\Gamma \vdash \lambda x:\tau. e : \tau \rightarrow \tau'} \text{T-ABS}$$

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'} \text{T-APP}$$

```
def tycheck(expr: Expr, tenv: Map[String, Type]): Type = expr match
  case Num(_) => NumT
  case Var(name) => tenv.getOrElse(name, error(s"unbound id: \${name}"))
  case Add(left, right) =>
    mustEqual(tycheck(left, tenv), NumT)
    mustEqual(tycheck(right, tenv), NumT)
    NumT
  case Abs(param, ty, body) =>
    val bodyTy = tycheck(body, tenv + (param -> ty))
    ArrowT(ty, bodyTy)
  case App(fun, arg) =>
    tycheck(fun, tenv) match
      case ArrowT(paramTy, retTy) =>
        mustEqual(tycheck(arg, tenv), paramTy)
        retTy
      case actual => error(s"arrow type expected, but got \${actual}")

def mustEqual(actual: Type, expected: Type): Unit =
  if (expected != actual) error(s"\${expected} expected, but got \${actual}")
```

$$\vdash (\lambda x : \text{num}. x + 40)(2)$$

$$\vdash (\lambda f : \text{num} \rightarrow \text{num}. f(2))(\lambda x : \text{num}. x) : \text{num}$$

$$\frac{\frac{\frac{\overline{\Gamma_0 \vdash x : \text{num}} \quad \overline{\Gamma_0 \vdash 40 : \text{num}}}{\Gamma_0 \vdash x + 40 : \text{num}}}{\vdash \lambda x : \text{num}. x + 40 : \text{num} \rightarrow \text{num}} \quad \overline{\vdash 2 : \text{num}}}{\vdash (\lambda x : \text{num}. x + 40)(2)}$$

where $\Gamma_0 = [x : \text{num}]$.

$$\vdash (\lambda f : \text{num} \rightarrow \text{num}. f(2))(\lambda x : \text{num}. x) : \text{num}$$

$$\frac{\frac{\frac{\overline{\Gamma_0 \vdash x : \text{num}} \quad \overline{\Gamma_0 \vdash 40 : \text{num}}}{\Gamma_0 \vdash x + 40 : \text{num}}}{\vdash \lambda x : \text{num}. x + 40 : \text{num} \rightarrow \text{num}} \quad \overline{\vdash 2 : \text{num}}}{\vdash (\lambda x : \text{num}. x + 40)(2)}$$

where $\Gamma_0 = [x : \text{num}]$.

$$\frac{\frac{\frac{\overline{\Gamma_0 \vdash f : \text{num} \rightarrow \text{num}} \quad \overline{\Gamma_0 \vdash 2 : \text{num}}}{\Gamma_0 \vdash f(2) : \text{num}}}{\vdash \lambda f : \text{num} \rightarrow \text{num}. f(2) : (\text{num} \rightarrow \text{num}) \rightarrow \text{num}} \quad \frac{\overline{\Gamma_1 \vdash x : \text{num}}}{\vdash \lambda x : \text{num}. x : \text{num} \rightarrow \text{num}}}{\vdash (\lambda f : \text{num} \rightarrow \text{num}. f(2))(\lambda x : \text{num}. x) : \text{num}}$$

where $\Gamma_0 = [f : \text{num} \rightarrow \text{num}]$ and $\Gamma_1 = [x : \text{num}]$.

1. Simply-Typed Lambda Calculus

Abstract Syntax

Dynamic Semantics

Static Semantics

2. Type Soundness

Type Soundness

Progress

Preservation

Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

Theorem (Type Soundness)

The STLC is type sound:

$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \nrightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$

Theorem (Type Soundness)

The STLC is type sound:

$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$

Lemma (Progress)

If an expression is well-typed, then it is either a value or it can take a small-step evaluation step:

$$(\vdash e : \tau) \implies (e \text{ is a value or } \exists e'. e \rightarrow e')$$

Theorem (Type Soundness)

The STLC is type sound:

$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$

Lemma (Progress)

If an expression is well-typed, then it is either a value or it can take a small-step evaluation step:

$$(\vdash e : \tau) \implies (e \text{ is a value or } \exists e'. e \rightarrow e')$$

Lemma (Preservation)

If a well-typed expression has a small-step evaluation step, then the resulting expression is also well-typed with the same type:

$$(\vdash e : \tau \wedge e \rightarrow e') \implies (\vdash e' : \tau)$$

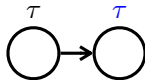
$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$



Progress



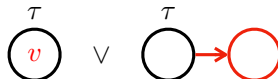
Preservation



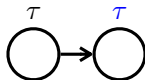
$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$



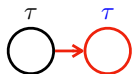
Progress



Preservation



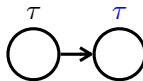
$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$



Progress



Preservation



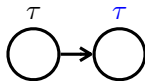
$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$



Progress



Preservation



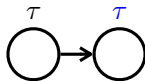
$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$



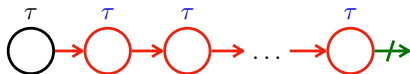
Progress



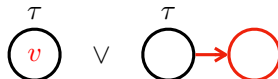
Preservation



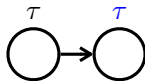
$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$



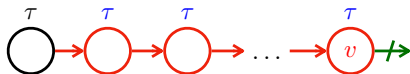
Progress



Preservation



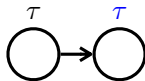
$$(\vdash e : \tau \wedge e \rightarrow^* e' \wedge e' \not\rightarrow) \implies (e' \text{ is a value and } \vdash e' : \tau)$$



Progress



Preservation



Lemma (Canonical Forms)

If a value v has a type τ (i.e., $\vdash v : \tau$), then:

- 1 *If $\tau = \text{num}$, then $v = n$ for some n .*
- 2 *If $\tau = \tau_1 \rightarrow \tau_2$, then $v = \lambda x : \tau_1. e$ for some x and e .*

Lemma (Canonical Forms)

If a value v has a type τ (i.e., $\vdash v : \tau$), then:

- 1 *If $\tau = \text{num}$, then $v = n$ for some n .*
- 2 *If $\tau = \tau_1 \rightarrow \tau_2$, then $v = \lambda x : \tau_1. e$ for some x and e .*

Using the canonical forms lemma, we can prove the progress lemma:

Lemma (Progress)

If an expression is well-typed, then it is either a value or it can take a small-step evaluation step:

$$(\vdash e : \tau) \implies (e \text{ is a value or } \exists e'. e \rightarrow e')$$

Let's prove the progress lemma by induction on the structure of typing rules.

- T-NUM and T-ABS cases are trivial since they are values.

$$\frac{}{\Gamma \vdash n : \mathbf{num}} \text{T-NUM} \qquad \frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \text{T-ABS}$$

- T-NUM and T-ABS cases are trivial since they are values.

$$\frac{}{\Gamma \vdash n : \mathbf{num}} \text{T-NUM} \qquad \frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \text{T-ABS}$$

- T-VAR case is impossible since variables with an empty typing environment cannot be well-typed.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \text{T-VAR}$$

- T-ADD case. $\vdash e_1 + e_2 : \text{num}$ implies $\vdash e_1 : \text{num}$ and $\vdash e_2 : \text{num}$.

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}} \text{ T-ADD}$$

- T-ADD case. $\vdash e_1 + e_2 : \text{num}$ implies $\vdash e_1 : \text{num}$ and $\vdash e_2 : \text{num}$.

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}} \text{ T-ADD}$$

By induction hypothesis, each e_i is either 1) a value (i.e., $e_i = v_i$) or 2) can take a step (i.e., $e_i \rightarrow e'_i$).

- T-ADD case. $\vdash e_1 + e_2 : \text{num}$ implies $\vdash e_1 : \text{num}$ and $\vdash e_2 : \text{num}$.

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}} \text{ T-ADD}$$

By induction hypothesis, each e_i is either 1) a value (i.e., $e_i = v_i$) or 2) can take a step (i.e., $e_i \rightarrow e'_i$).

- If e_1 is not a value, by the CONTEXT rule,

$$e_1 + e_2 \rightarrow e'_1 + e_2 \quad (E = [\cdot] + e_2)$$

- T-ADD case. $\vdash e_1 + e_2 : \text{num}$ implies $\vdash e_1 : \text{num}$ and $\vdash e_2 : \text{num}$.

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}} \text{ T-ADD}$$

By induction hypothesis, each e_i is either 1) a value (i.e., $e_i = v_i$) or 2) can take a step (i.e., $e_i \rightarrow e'_i$).

- If e_1 is not a value, by the CONTEXT rule,

$$e_1 + e_2 \rightarrow e'_1 + e_2 \quad (E = [\cdot] + e_2)$$

- If e_1 is a value but e_2 is not a value, by the CONTEXT rule,

$$v_1 + e_2 \rightarrow v_1 + e'_2 \quad (E = v_1 + [\cdot])$$

- T-ADD case. $\vdash e_1 + e_2 : \text{num}$ implies $\vdash e_1 : \text{num}$ and $\vdash e_2 : \text{num}$.

$$\frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}} \text{ T-ADD}$$

By induction hypothesis, each e_i is either 1) a value (i.e., $e_i = v_i$) or 2) can take a step (i.e., $e_i \rightarrow e'_i$).

- If e_1 is not a value, by the CONTEXT rule,

$$e_1 + e_2 \rightarrow e'_1 + e_2 \quad (E = [\cdot] + e_2)$$

- If e_1 is a value but e_2 is not a value, by the CONTEXT rule,

$$v_1 + e_2 \rightarrow v_1 + e'_2 \quad (E = v_1 + [\cdot])$$

- If e_1 and e_2 are both values, then by the canonical forms lemma and $\vdash e_i : \text{num}$, they must be numbers ($e_1 = n_1$ and $e_2 = n_2$). So, we can take a step using the ADD rule:

$$n_1 + n_2 \rightarrow n \quad (n = n_1 + n_2)$$

- T-APP case. $\vdash e_1 e_2 : \tau$ implies $\vdash e_1 : \tau' \rightarrow \tau$ and $\vdash e_2 : \tau'$.

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \text{ T-APP}$$

- T-APP case. $\vdash e_1 e_2 : \tau$ implies $\vdash e_1 : \tau' \rightarrow \tau$ and $\vdash e_2 : \tau'$.

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \text{ T-APP}$$

By induction hypothesis, each e_i is either 1) a value (i.e., $e_i = v_i$) or 2) can take a step (i.e., $e_i \rightarrow e'_i$).

- If e_1 is not a value, by the CONTEXT rule,

$$e_1 e_2 \rightarrow e'_1(e_2) \quad (E = [\cdot] e_2)$$

- If e_1 is a value but e_2 is not a value, by the CONTEXT rule,

$$v_1 e_2 \rightarrow v e'_2 \quad (E = v [\cdot])$$

- T-APP case. $\vdash e_1 e_2 : \tau$ implies $\vdash e_1 : \tau' \rightarrow \tau$ and $\vdash e_2 : \tau'$.

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \text{ T-APP}$$

By induction hypothesis, each e_i is either 1) a value (i.e., $e_i = v_i$) or 2) can take a step (i.e., $e_i \rightarrow e'_i$).

- If e_1 is not a value, by the CONTEXT rule,

$$e_1 e_2 \rightarrow e'_1(e_2) \quad (E = [\cdot] e_2)$$

- If e_1 is a value but e_2 is not a value, by the CONTEXT rule,

$$v_1 e_2 \rightarrow v e'_2 \quad (E = v [\cdot])$$

- If e_1 and e_2 are both values, then by the canonical forms lemma and $\vdash e_1 : \tau' \rightarrow \tau$, e_1 must be a lambda abstraction (i.e., $e_1 = \lambda x : \tau'. e$). So, we can take a step using the β -REDUCE rule:

$$(\lambda x : \tau'. e)(v) \rightarrow e[x \mapsto v]$$

Lemma (Substitution)

If $[x : \tau] \vdash e : \tau'$ and $\vdash v : \tau$, then $\vdash e[x \mapsto v] : \tau'$.

Lemma (Substitution)

If $[x : \tau] \vdash e : \tau'$ and $\vdash v : \tau$, then $\vdash e[x \mapsto v] : \tau'$.

Lemma (Context)

If $\vdash E[e] : \tau$ and $\vdash e : \tau'$, then $\vdash E[e'] : \tau$ for any e' such that $\vdash e' : \tau'$.

Lemma (Substitution)

If $[x : \tau] \vdash e : \tau'$ and $\vdash v : \tau$, then $\vdash e[x \mapsto v] : \tau'$.

Lemma (Context)

If $\vdash E[e] : \tau$ and $\vdash e : \tau'$, then $\vdash E[e'] : \tau$ for any e' such that $\vdash e' : \tau'$.

We will skip the detailed proof of above lemmas, but they can be proved by induction on the structure of typing rules.

Lemma (Substitution)

If $[x : \tau] \vdash e : \tau'$ and $\vdash v : \tau$, then $\vdash e[x \mapsto v] : \tau'$.

Lemma (Context)

If $\vdash E[e] : \tau$ and $\vdash e : \tau'$, then $\vdash E[e'] : \tau$ for any e' such that $\vdash e' : \tau'$.

We will skip the detailed proof of above lemmas, but they can be proved by induction on the structure of typing rules.

Lemma (Preservation)

If a well-typed expression has a small-step evaluation step, then the resulting expression is also well-typed with the same type:

$$(\vdash e : \tau \wedge e \rightarrow e') \implies (\vdash e' : \tau)$$

- ADD case.

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n} \text{ADD}$$

- ADD case.

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n} \text{ ADD}$$

$n_1 + n_2$ has type `num` by the T-ADD and T-NUM rules:

$$\frac{\frac{}{\vdash n_1 : \text{num}} \text{ T-NUM} \quad \frac{}{\vdash n_2 : \text{num}} \text{ T-NUM}}{\vdash n_1 + n_2 : \text{num}} \text{ T-ADD}$$

- ADD case.

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n} \text{ ADD}$$

$n_1 + n_2$ has type `num` by the T-ADD and T-NUM rules:

$$\frac{\frac{}{\vdash n_1 : \text{num}} \text{ T-NUM} \quad \frac{}{\vdash n_2 : \text{num}} \text{ T-NUM}}{\vdash n_1 + n_2 : \text{num}} \text{ T-ADD}}$$

Similarly, n has type `num` by the T-NUM rule:

$$\frac{}{\vdash n : \text{num}} \text{ T-NUM}$$

So, the preservation property holds by preserving the type `num`.

- β -REDUCE case.

$$\frac{}{(\lambda x:\tau. e)(v) \rightarrow e[x \mapsto v]} \beta\text{-REDUCE}$$

- β -REDUCE case.

$$\frac{}{(\lambda x:\tau. e)(v) \rightarrow e[x \mapsto v]} \beta\text{-REDUCE}$$

$(\lambda x:\tau. e)(v)$ is well-typed. Let's assume that it has type τ' .

- β -REDUCE case.

$$\overline{(\lambda x:\tau. e)(v) \rightarrow e[x \mapsto v]} \quad \beta\text{-REDUCE}$$

$(\lambda x:\tau. e)(v)$ is well-typed. Let's assume that it has type τ' .

Then, we have the following typing derivation:

$$\frac{\frac{[x:\tau] \vdash e:\tau'}{\vdash \lambda x:\tau. e:\tau \rightarrow \tau'} \text{T-ABS} \quad \frac{\dots}{\vdash v:\tau}}{\vdash (\lambda x:\tau. e)(v):\tau'} \text{T-APP}$$

- β -REDUCE case.

$$\frac{}{(\lambda x:\tau. e)(v) \rightarrow e[x \mapsto v]} \beta\text{-REDUCE}$$

$(\lambda x:\tau. e)(v)$ is well-typed. Let's assume that it has type τ' .

Then, we have the following typing derivation:

$$\frac{\frac{[x:\tau] \vdash e:\tau'}{\vdash \lambda x:\tau. e:\tau \rightarrow \tau'} \text{T-ABS} \quad \frac{\dots}{\vdash v:\tau}}{\vdash (\lambda x:\tau. e)(v):\tau'} \text{T-APP}$$

By the substitution lemma, $[x:\tau] \vdash e:\tau'$ and $\vdash v:\tau$ imply:

$$\vdash e[x \mapsto v]:\tau'$$

So, the preservation property holds by preserving the type τ' .

- CONTEXT case.

$$\frac{e_1 \rightarrow e_2}{E[e_1] \rightarrow E[e_2]} \text{CONTEXT}$$

$$E ::= [\cdot] \mid E e \mid v E \mid E + e \mid v + E$$

- CONTEXT case.

$$\frac{e_1 \rightarrow e_2}{E[e_1] \rightarrow E[e_2]} \text{CONTEXT}$$

$$E ::= [\cdot] \mid E e \mid v E \mid E + e \mid v + E$$

$E[e_1]$ is well-typed. Let's assume that it has type τ .

- CONTEXT case.

$$\frac{e_1 \rightarrow e_2}{E[e_1] \rightarrow E[e_2]} \text{CONTEXT}$$

$$E ::= [\cdot] \mid E e \mid v E \mid E + e \mid v + E$$

$E[e_1]$ is well-typed. Let's assume that it has type τ .

By the induction on the structure of evaluation contexts, we can show that e_1 is well-typed with some type τ_1 (i.e., $\vdash e_1 : \tau_1$).

- CONTEXT case.

$$\frac{e_1 \rightarrow e_2}{E[e_1] \rightarrow E[e_2]} \text{CONTEXT}$$

$$E ::= [\cdot] \mid E e \mid v E \mid E + e \mid v + E$$

$E[e_1]$ is well-typed. Let's assume that it has type τ .

By the induction on the structure of evaluation contexts, we can show that e_1 is well-typed with some type τ_1 (i.e., $\vdash e_1 : \tau_1$).

By the induction hypothesis and $e_1 \rightarrow e_2$, we have $\vdash e_2 : \tau_1$.

- CONTEXT case.

$$\frac{e_1 \rightarrow e_2}{E[e_1] \rightarrow E[e_2]} \text{CONTEXT}$$

$$E ::= [\cdot] \mid E e \mid v E \mid E + e \mid v + E$$

$E[e_1]$ is well-typed. Let's assume that it has type τ .

By the induction on the structure of evaluation contexts, we can show that e_1 is well-typed with some type τ_1 (i.e., $\vdash e_1 : \tau_1$).

By the induction hypothesis and $e_1 \rightarrow e_2$, we have $\vdash e_2 : \tau_1$.

By the context lemma, $\vdash E[e_1] : \tau$ and $\vdash e_1 : \tau_1$ imply:

$$\vdash E[e_2] : \tau$$

So, the preservation property holds by preserving the type τ .

- Type Extensions

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>