Lecture 23. Type Systems III

1. Quiz: Write Typing Rules
2. More on Subtyping:
   - 2.1. Definition, Use Case (Polymorphism).
   - 2.2. Subtyping on Tuples
   - 2.3. Subtyping on Functions
   - 2.4. Theorems

$$\frac{\Gamma \vdash e : t \qquad \vdash t <: t'}{\Gamma \vdash e : t'}$$

Re

3. Theorem on Type Systems.
4. Rust: Borrowing & life time & lexical Scoping

```
r1 { let x = 0;
r2 { let y = x;
r3 { let z = &x;   // &i32
   }
```

F^b i32 Ref

$r^3 \sqsubseteq r^2 \sqsubseteq r^1$

$$\Gamma ; R \vdash e : t$$

$$\{x : i32\} ; \{x : r^1\} \vdash \text{let } z = \&x ; z : \langle i32, r^3 \rangle$$

$$\Gamma \vdash e : \tau \dashv \Gamma'$$

- Γ: previous typing context
- e: expr
- τ: type
- Γ': resulting typing context

5. Final Review:

a. Syntax & Semantics $e \Rightarrow v$ ($e ; v$)

b. Language Extensions and features.

b.1). Simple ones $e \Rightarrow v$.

tuple, variants, records, exceptions

b.2). Complicated ones (Altering semantics)

State $\langle e, \sigma_1 \rangle \Rightarrow \langle v, \sigma_2 \rangle$

Concurrency $\begin{cases} G_0 \to G_1 \to \dots \\ e \overset{S}{\Rightarrow} v \end{cases}$

Type System $\Gamma \vdash e : t$

b.3). Program Transformations.

Tuple / Structs $\to$ Lambda Calculus

Object oriented programming $\to$ $F^b R$

Key Takeaways:

$(\lambda x.x)\ y \to_\beta y$

$\lambda f.\lambda x.f\ (f\ x)$

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ (\to I)$$

Programming Languages Course — Key Takeaways

From mathematical foundations to language design, transfer of ideas, curiosity, and correctness in the age of AI

$e:\tau \Rightarrow e':\tau$

$\langle e,\sigma\rangle \longrightarrow \langle e',\sigma'\rangle$

$\forall x.\,P(x) \Rightarrow Q(x)$

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)

type 'a option =
  | None
  | Some of 'a
```

1. First Principles

Programming languages are fundamentally mathematical constructs. Once you understand the core mathematical ideas, syntax, semantics, concurrency, objects, state, memory, and networking become elegant extensions of the same formal foundations.

Lambda Calculus

λ

$(\lambda x.x)\ y \to_\beta y$

Inference Rule (Typing)

$$\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ (\to I)$$

Semantic Judgment (Big-Step)

$\langle e,\sigma\rangle \Downarrow v$

Abstract Syntax Tree (AST)

App, Lam, Var, Var, x, y

Equational Reasoning

let id = $\lambda x.x$ in id v

$= (\lambda x.x)\ v$

$\to_\beta v$

2. Languages Can Be Designed

In this course, we built our own programming language and its formal semantics. Languages are not mysterious artifacts — they are designed by people. You can create your own languages too, especially domain-specific languages for particular tasks.

Build a Toy Language

Values · Types · Expressions · Evaluation · Environment · State

MyLang Core

Grammar (EBNF)

```
e ::= n
   | x
   | λx.e
   | e e
   | let x = e in e
   | if e then e else e
```

Parser Sketch

Source → Lexer (Tokens) [...] → Parser (AST)

Operational Semantics (Small-Step)

$\langle e,\sigma\rangle \longrightarrow \langle e',\sigma'\rangle$

$\langle (\lambda x.e)\ v, \sigma\rangle \longrightarrow \langle e[v/x], \sigma\rangle$

$\langle \text{if true then } e_1 \text{ else } e_2, \sigma\rangle \longrightarrow \langle e_1, \sigma\rangle$

$\langle x, \sigma\rangle \longrightarrow \langle \sigma(x), \sigma\rangle$

DSL Example: QueryLang

from Users → where age > 18 → select name → execute

Compiled to an efficient plan

3. Ideas Transfer Across Languages

When you encounter Rust, Julia, Go, Haskell, or OCaml, do not be intimidated. Use the concepts from this class to understand them. Most new languages are rephrasings or extensions of ideas you have already seen before.

Rust

```
fn add(x: i32, y: i32)
  -> i32 { x + y }

struct Point {x: i32,
  y: i32}
```

Julia

```
add(x::Int, y::Int) =
  x + y

struct Point
  x::Int; y::Int
end
```

Shared Core

- Types
- Functions
- Data Structures
- Control Flow
- Effects & State
- Semantics

Go

```
func add(x, y int) int {
  return x + y
}

type Point struct {
  X, Y int
}
```

Haskell

```
add x y = x + y

data Point = Point
  { x :: Int
  , y :: Int }
  deriving (Show)
```

OCaml

```
let add x y = x + y

type point = { x: int; y: int }
```

Different surface syntax, same deep ideas.
Learn once, apply everywhere.

4. Embrace Exploration

Do not be afraid to play with programming languages, learn unfamiliar ones, and explore new paradigms. New languages are opportunities to deepen your understanding, not obstacles to avoid.

Functional: Haskell, OCaml, F#

Systems: Rust, C, Zig

Data Science: Julia, R, Python

Concurrent: Go, Erlang, Scala

Scripting: Lua, Python, JavaScript

Logic: Prolog, Datalog

Experimental: Idris, Lean, Elm

Read · Build · Tinker

Compare · Ask Why

Measure · Reflect

Playground Sandbox

5. What Makes a Program Correct?

A central lesson of programming languages is understanding correctness. In the era of AI-assisted coding, this matters even more. Correctness may involve complatadion success, type checking, functional correctness, formal correctness, observational equivalence, and safety properties.

- **Compilation Success** — Program compiles without errors. (Compiler ✓)
- **Type Safety** — Types are consistent; no type errors. (Type Checker ✓)
- **Functional Correctness** — Produces the specified results for all inputs. (Tests / Specs ✓)
- **Formal Correctness** — Proved with logic, proofs, or theorems. (Proof Assistant: $\forall x.\ P(x) \Rightarrow Q(x)$ ✓)
- **Observational Equivalence** — Indistinguishable in all observable contexts. (Equivalance Check ✓)
- **Safety Properties** — Memory safe, no crashes, data-race free, etc. (Verifier / Analyzer ✓)

AI Assistant ("How can I help?") → Generated Code

```
def add(x, y):
  return x + y
```

→ Verification Toolchain

AI can help write code. You ensure it is correct.

$e \Downarrow v$

Foundations

Design

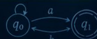
$\lambda x.\ (x\ x)$

Transfer

Exploration

$\pi_1\ \langle v_1, v_2\rangle = v_1$

Correctness



Lecture 23. Type System III

Subtypes

$t_1 <: t_2$

Polymorphism

$I \equiv \lambda x.\ x$

get_x = Fun p : {x : Int} → p.x (t_2)

get_x {x = 1 ; y = 2} : {x : Int ; y : Int}

get_x {x = 1} {x : Int}

of different types

$t_1 <: t_2$

value of t_1 can be used when t_2 is expected

{x : Int} <: {x : Int}

{x : Int , y : Int} <: {x : Int}

Subtypes between Functions

$A \to B <: C \to D$

Iff ⟺

$C <: A$ contra-variant

$B <: D$ co-variant

Func type's subtype related to Input type's subtype

$t ::= \text{Animal} \mid \text{Dog}$

$\text{Dog} <: \text{Animal}$

subtype?

supertype?

$\text{Dog} \to \text{Int} :> \text{Animal} \to \text{Int}$

$\text{Int} \to \text{Dog} <: \text{Int} \to \text{Animal}$

$\text{Int} \not<: \text{Bool}$ ($\neq$)

Type Systems:

$e \Rightarrow v$

Op Sem \ Type	Accept	Reject
Succeed		$FN \neq \emptyset$
Fail	$FP \neq \emptyset$	

Small step $e \rightsquigarrow e'$, $e \neq e'$ \ Type	Accept	Reject
Diverge / Value Non-Stuck	✓	$FN \neq \emptyset$
Stuck	$\emptyset$	✓

If True Then {} Else False

Theorem: If e, $\{\} \vdash e : t$
Then evaluating e will not stuck

$$[\text{Left Step Add}] \quad \frac{e_1 \rightsquigarrow e_1'}{e_1 + e_2 \rightsquigarrow e_1' + e_2}$$

$$[\text{Right} \ldots] \quad \frac{e_2 \rightsquigarrow e_2'}{e_1 + e_2 \rightsquigarrow e_1 + e_2'}$$

$$[\text{Add Base}] \quad \frac{i_1 \text{ is int} \qquad i_2 \text{ is int}}{\bar{i}_1 + \bar{i}_2 \rightarrow \bar{i}' \text{ when } i' \text{ is integer sum of } i_1 \text{ and } i_2}$$

Subtyping

$$\frac{\Gamma \vdash e : t \qquad t <: t'}{\Gamma \vdash e : t'}$$

$$\frac{t <: t' \quad \Gamma \vdash e : t}{\Gamma \vdash e : t'}$$

evaluate e will not get stuck when t' is expected

$F^b \qquad \Gamma \vdash e : t$

Δ

Rust

$$\Gamma \vdash e : t \dashv \Gamma'$$

$$\frac{x : t \in \Gamma \qquad \Gamma' = \Gamma / \{x : t\} \cup \{y : t\}}{\Gamma \vdash \text{let } y = x; : () \dashv \Gamma'}$$

ctx before

unit

ctx after

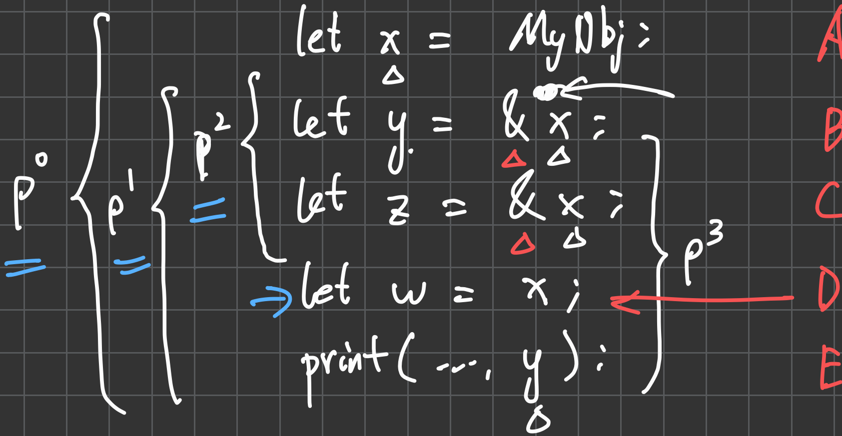
$$\frac{x : t \in \Gamma}{\Gamma \vdash x + x \dashv \Gamma'}$$

$$\frac{\Gamma_1 \vdash e_1 : t_1 \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : t_2 \dashv \Gamma_3 \qquad \Gamma_3 \vdash add(e_1, e_2) : t_3 \dashv \Gamma_4}{\Gamma_1 \vdash e_1 + e_2 : t_3 \dashv \Gamma_4}$$

Rust

$t ::= i32 \mid i64 \mid MyObject \mid \ldots$

$\mid \&^{\rho}\, t$ ← region / life time



Question: Does x live all the way till E

$\{ y : \&^{\rho^1} MyObj \} \vdash$ let w = x $\dashv$