Lecture 4: Simply Typed Lambda Calculus

Review

Lambda Calculus recap

- Grammar: $\Lambda = V \mid \Lambda\,\Lambda \mid \lambda V.\,\Lambda$
- Only computational rule — β-reduction:

$$(\lambda x.\, N)\, M \;\to_\beta\; N[x := M]$$

- **Church–Rosser theorem:** if $M \twoheadrightarrow_\beta M'$ and $M \twoheadrightarrow_\beta N'$ then there exists P with $M' \twoheadrightarrow_\beta P$ and $N' \twoheadrightarrow_\beta P$.

Consequence of Church–Rosser

- Every λ-term either reduces to a **unique** β-**normal form** (cannot be simplified further), or can be β-reduced indefinitely.
- The non-trivial part: the β-normal form is **unique** when it exists.
- We view it as the outcome of the computation.

Non-termination: Ω

- A λ-term that loops forever:

$$\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

- The analogue of `while(1) { }`.
- A useless program: wastes resources, never returns.
- **Goal:** design a language where such programs cannot be written.

The halting problem

Theorem

There is no algorithm that can decide, for all programs, whether a given program will terminate.

Since λ-calculus is as powerful as Turing machines:

Theorem

There is no algorithm that can decide, for all λ-terms, whether a given λ-term can be reduced to β-normal form.

Getting around the halting problem

- Something feels off... By inspecting code you can *usually* figure out if a program terminates.
- The key word is **for all programs**. Maybe we only ever look at a subset where termination *can* be determined.
- **Refined goal:** specify a subset of λ-calculus such that membership is decidable and guarantees termination.

Strong normalization

- We aim for something stricter: a subset where every term is **strongly normalizing**.
- *Strongly normalizing* means: **every** reduction path is finite, regardless of which redex we choose.
- The idea: introduce **types** as a decidable, syntactic criterion checked *before* the program runs, that implies termination.

Introducing Types

The idea behind types

- Each variable x is associated with a type. If x has type A we write $x : A$.
- Functions accept inputs of one type and return outputs of another. We write $f : A \to B$.
- **Key restriction:** an application $M\,N$ is only allowed when M has a function type.

Why Ω cannot be typed

- What type would $\lambda x.\, x\, x$ have?
- Since we see $x\, x$, the variable x must be a function: say $x : A \to B$.
- But x is also the argument, so we need $A = A \to B$.
- This is circular: $A \to B$ is strictly longer than A. No finite type satisfies this!
- Therefore $\lambda x.\, x\, x$ is **ill-typed**, and Ω cannot be written.

Types as dimensional analysis

- Think of types as dimensions in physics.
- You cannot add meters to seconds. Similarly, you cannot apply a function of type $A \to B$ to an argument of type $C \neq A$.
- Just as dimensional analysis sometimes lets you *guess* the correct formula, types will help us *guess* the correct program.

Typing Church Numerals

Church numerals

- Recall $\overline{2} = \lambda f.\, \lambda x.\, f\,(f\,x)$.
- If $x : \alpha$ then $f : \alpha \to \beta$. Since f is also applied to $f\,x$, we need $\beta = \alpha$.
- So $f : \alpha \to \alpha$, $x : \alpha$, and:

$$\overline{2} : (\alpha \to \alpha) \to \alpha \to \alpha$$

All Church numerals have the same type

$$\overline{n} = \lambda f.\, \lambda x.\, f^n\, x \;:\; (\alpha \to \alpha) \to \alpha \to \alpha$$

- Shorthand: $\mathsf{nat} := (\alpha \to \alpha) \to \alpha \to \alpha$.
- What is α? A fixed but arbitrary type variable.
- Later: **polymorphism** (System F) will let us quantify over α explicitly.

Typing successor

- $\mathsf{succ} = \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\,f\,x)$
- Type: $\mathsf{nat} \to (\beta \to \beta) \to \beta \to \beta$
- Since $\mathsf{nat} = (\alpha \to \alpha) \to \alpha \to \alpha$, we are forced to pick $\beta = \alpha$:

$$\mathsf{succ} : \mathsf{nat} \to \mathsf{nat} \quad \checkmark$$

Types guide the program: addition

- We want $+ \ : \ \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$.
- Expanding, the function must start as:

  $$+ \ := \ \lambda n.\, \lambda m.\, \lambda f.\, \lambda x.\, (\ldots)$$

  where the body has type α.
- We have $n, m : \mathsf{nat}$, $f : \alpha \to \alpha$, $x : \alpha$.
- Types leave very few valid compositions!

Two guesses, both valid

- **Guess 1:**

$$n\ f\ (m\ f\ x) \quad \longrightarrow \quad \text{addition } (n + m)$$

- **Guess 2:**

$$n\ (m\ f)\ x \quad \longrightarrow \quad \text{multiplication } (n \times m)$$

- Types restrict the space of programs and help us write them.
- Both compositions are well-typed; both are sensible programs.

Associativity conventions

- **Types** are right-associative:

$$\alpha \to \alpha \to \alpha \quad \text{means} \quad \alpha \to (\alpha \to \alpha)$$

- **Terms** are left-associative:

$$f\ a\ b \quad \text{means} \quad (f\ a)\ b$$

- These dual conventions are exactly **currying**: if $f : A \to B \to C$ then

$$f : A \to (B \to C), \quad f\ a : B \to C, \quad f\ a\ b : C$$

Formal Definition

Grammar of types

- Fix type variables $\mathbb{V} = \{\alpha, \beta, \gamma, \ldots\}$.
- The set of **simple types**:

$$\mathbb{T} \ ::= \ \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$$

- Examples: α, $\alpha \to \beta$, $(\alpha \to \alpha) \to \alpha \to \alpha$.

Grammar of typed terms

- Abstractions must declare the type of their argument:

$$\Lambda_{\mathbb{T}} \;=\; V \;\mid\; \Lambda_{\mathbb{T}}\,\Lambda_{\mathbb{T}} \;\mid\; \lambda V : \mathbb{T}.\, \Lambda_{\mathbb{T}}$$

- Just satisfying this grammar is *not enough*. A term must also obey **typing rules** to be valid.
- This extra enforcement is what buys us the termination guarantee.

Typing rules, informally

- **Rule 1:** If x has type σ, then we can assert $x : \sigma$.
- **Rule 2:** If M has type $\sigma \rightarrow \tau$ and N has type σ, then $M\,N$ is a valid expression and has type τ.
- **Rule 3:** If x has type σ and M has type τ, then $\lambda x : \sigma.\, M$ is a valid expression and has type $\sigma \rightarrow \tau$.

Why these rules?

- They express exactly how types and terms are interrelated.
- Type-checking can always be performed in finite time, *before* the program runs.
- Only terms that can be built using these rules are declared valid. This enforcement is what will buy us the termination guarantee.

Typing contexts

- A **context** Γ is a finite list of type assignments:

$$\Gamma = x_1 : \sigma_1,\ x_2 : \sigma_2,\ \ldots,\ x_n : \sigma_n$$

- We write the following to express that, under assumptions Γ, the term M has type τ:

$$\Gamma \vdash M : \tau$$

- This notation is called a **sequent**. The turnstile $\vdash$ separates what we assume (left) from what we conclude (right).

What is a context?

- A context Γ is a record of **what we currently know**: it lists the variables that are in scope and what type each one has.
- When we write $\Gamma \vdash M : \tau$, we are saying: "given that the variables have the types listed in Γ, we can conclude that M has type τ."
- This is a general logical notion. In logic, you might write $\Gamma \vdash P$ to mean "from the hypotheses Γ, we can prove P." Here it is the same idea, but for types.

Context: a concrete example

- Inside the body of $\lambda x : \alpha.\, \lambda y : \beta.\, (\ldots)$, the variables in scope are x and y, so $\Gamma = x : \alpha,\ y : \beta$.
- If we encounter a variable z that is not in Γ, we cannot assign it a type — the expression is invalid.
- If you like, you can think of Γ as what a compiler **keeps in mind** while type-checking: it maintains a table of which variables are available and what their types are. But the concept is purely mathematical — it is just the list of assumptions we are allowed to use.

What does the sequent notation mean?

- The typing rules will have the form:

$$\frac{\text{statement(s) above the line}}{\text{statement below the line}}$$

- This simply reads: **if** the state of affairs above the line holds, **then** the state of affairs below the line also holds.
- For example, a rule might look like:

$$\frac{\Gamma \vdash f : \alpha \to \alpha \qquad \Gamma \vdash x : \alpha}{\Gamma \vdash f\ x : \alpha}$$

- This says: if from our current knowledge Γ we can deduce that f has type $\alpha \to \alpha$, and from the same knowledge Γ we can deduce that x has type α, then from Γ we can also deduce that $f\ x$ has type α.

Deduction trees

- We can **chain** these rules together. Each statement above the line might itself be the conclusion of another rule, and so on.
- This produces a **deduction tree**: we start from things we know (what's in the context) and work our way down to the conclusion.
- If we can build such a tree, the term is **well-typed**. If we cannot, the term is rejected.

How contexts change

- As we move through a term, the context can **grow**: when we enter the body of $\lambda x : \sigma. (\ldots)$, we add $x : \sigma$ to what we know, because x is now in scope.
- The context can also **shrink**: once we leave the body of a λ-abstraction, x is no longer in scope, so we remove it. We only keep track of what is currently relevant.

The three typing rules

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ (\textsc{App})$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.\, M : \sigma \to \tau}\ (\textsc{Abs})$$

Reading the rules

- **(Var):** If $x : \sigma$ is listed in Γ, then from Γ we can conclude $x : \sigma$. We simply look up what we already know.
- **(App):** If from Γ we can deduce that M has type $\sigma \to \tau$, and from Γ we can deduce that N has type σ, then from Γ we can also deduce that $M\,N$ has type τ. This enforces type compatibility — like dimensional analysis.
- **(Abs):** If from the *extended* context $\Gamma, x : \sigma$ we can deduce that the body M has type τ, then from Γ alone we can deduce that $\lambda x : \sigma.\, M$ has type $\sigma \to \tau$. The context **grows** above the line and **shrinks** below it — exactly as we described.

Example: deriving $\lambda x : \alpha . x \; : \; \alpha \to \alpha$

- Let us build a deduction tree for a simple term: the identity function $\lambda x : \alpha . x$.
- We start from the bottom: we want to conclude $\vdash \lambda x : \alpha . x : \alpha \to \alpha$. This is a λ-abstraction, so we use (Abs). The rule requires us to show $x : \alpha \vdash x : \alpha$, with $x : \alpha$ added to the context.
- But $x : \alpha$ is in the context, so (Var) applies. We are done:

$$\dfrac{\dfrac{(x : \alpha) \in \{x : \alpha\}}{x : \alpha \vdash x : \alpha}\ \textsc{Var}}{\vdash \lambda x : \alpha . x : \alpha \to \alpha}\ \textsc{Abs}$$

Notice how the context **grew** (we added $x : \alpha$ above the line in Abs) and then **shrank** (below the line, x is gone).

Example: deriving $\lambda f.\, \lambda x.\, f\, x$

- A slightly larger example. Let $\Gamma = f : \alpha \to \alpha,\ x : \alpha$.

$$\dfrac{\dfrac{\dfrac{\dfrac{(f : \alpha \to \alpha) \in \Gamma}{\Gamma \vdash f : \alpha \to \alpha}\ \textsc{Var} \qquad \dfrac{(x : \alpha) \in \Gamma}{\Gamma \vdash x : \alpha}\ \textsc{Var}}{\Gamma \vdash f\, x : \alpha}\ \textsc{App}}{f : \alpha \to \alpha \vdash \lambda x : \alpha.\, f\, x : \alpha \to \alpha}\ \textsc{Abs}}{\vdash \lambda f : \alpha \to \alpha.\, \lambda x : \alpha.\, f\, x : (\alpha \to \alpha) \to \alpha \to \alpha}\ \textsc{Abs}$$

All three rules appear: (Var) to look things up, (App) to check the application $f\, x$, and (Abs) twice to close off the two λs.

The workflow

- We could check more complex expressions the same way — like an iseven function applied to a Church numeral. The process is always the same: build a deduction tree using the three rules.
- If we succeed, the term is accepted. We *expect* that this gives us a guarantee about the execution of the program — namely, that it will terminate.
- But we have not proven this yet! This is the main result about the simply typed λ-calculus, and we state it now.

The payoff

Theorem (Strong Normalization)

Every well-typed term in the simply typed λ-calculus is strongly normalizing: every reduction sequence is finite.

- This is the theorem that justifies everything we have built. If we can construct a valid deduction tree for a term, we are **guaranteed** that every reduction path terminates.
- The types have then served their purpose. We can erase them and execute the program using ordinary β-reduction, knowing that the computation will finish.
- **Price:** the simply typed λ-calculus is *not* Turing complete. We will recover expressiveness with richer type systems later.

Sneak Peek: Types as Logic

Erase the terms

- Look at our three typing rules again, but **forget the terms**. Keep only the types and the context:

$$\frac{A \in \Gamma}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

- Read $A \to B$ as A *implies* B, and $\Gamma \vdash A$ as *from hypotheses* Γ, *we can prove* A.

These are the rules of logic

- The first rule says: if A is one of our hypotheses, we can conclude A.
- The second says: if we can prove $A \to B$ and we can prove A, then we can prove B. This is the classical rule: if A implies B, and A holds, then B holds.
- The third says: if by assuming A we can prove B, then we can conclude $A \to B$.
- These are the rules of **propositional logic** restricted to implication. The typing rules and the logic rules are *the same rules*.

Programs are proofs

- Under this reading, every well-typed λ-term corresponds to a proof, and every proof corresponds to a λ-term.

Type theory	**Logic**
Type σ	Proposition σ
Term $M : \sigma$	Proof of σ
$\sigma \to \tau$	"σ implies τ"
Application $M\ N$	From "A implies B" and A, conclude B
$\lambda x : \sigma.\, M$	Assume σ, prove τ
Context Γ	Hypotheses
β-reduction	Proof simplification

Example: A implies A

- The statement *if A then A* is trivially true. Can we prove it?

$$\dfrac{\dfrac{A \in \{A\}}{A \vdash A}}{\vdash A \to A}$$

- **Corresponding program:** $\lambda x : A.\, x$ (the identity function).
- The identity function **is** the proof that A implies A.

Example: from $A \to B$ and A, conclude B

Let $\Gamma = A \to B,\ A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{A \to B \in \Gamma}{\Gamma \vdash A \to B} \qquad \dfrac{A \in \Gamma}{\Gamma \vdash A}}{\Gamma \vdash B}}{A \to B \vdash A \to B}}{\vdash (A \to B) \to A \to B}$$

Program: $\lambda f : A \to B.\, \lambda x : A.\, f\, x$ (function application).

The punchline

- Every well-typed λ-term **is** a proof.
- Every proof in this logic **is** a program.
- β-reduction **is** proof simplification.
- Strong normalization **is** the statement that every proof can be reduced to a simplest form.

This correspondence between types and logic is the central theme of this course. Everything we do from here on will be elaborations of this idea.