编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao，Di Wang
赵海燕，王迪

Peking University, Spring Term 2025

Chapter 9:
Simply Typed Lambda-Calculus

Function Types

The Typing Relation

Properties of Typing

The Curry-Howard Correspondence

Erasure and Typability

The simply typed lambda-calculus

- The system we are about to define is commonly called the *simply typed lambda-calculus*, $\lambda_{\rightarrow}$, for short.
- Talking about $\lambda_{\rightarrow}$, we always begin with *some set of "base types"*, unlike the *untyped lambda-calculus*, the "pure" form of $\lambda_{\rightarrow}$ (with no primitive values or operations) is not very interesting
  - Strictly speaking, there are ***many variants*** of $\lambda_{\rightarrow}$, depending on *the choice of* ***base types***.
  - For now, we'll work with *a variant constructed over the booleans*.

Untyped lambda-calculus with booleans

t	::=		*terms*
		x	*variable*
		λx.t	*abstraction*
		t t	*application*
		true	*constant true*
		false	*constant false*
		if t then t else t	*conditional*

v	::=		*values*
		λx.t	*abstraction value*
		true	*true value*
		false	*false value*

Function Types

- $T_1 \longrightarrow T_2$
  - *classifying functions* that expect arguments of type T1 and return results of type T2.

- $\longrightarrow$: type constructor is *right-associative*, e.g.,

  $T_1 \longrightarrow T_2 \longrightarrow T_3$ stands for $T_1 \longrightarrow (T_2 \longrightarrow T_3)$

- Let's consider *Booleans* with lambda calculus

T ::=	types :
Bool	type of booleans
$\mathrm{T} \longrightarrow \mathrm{T}$	type of functions

- Examples
  - $\mathrm{Bool} \longrightarrow \mathrm{Bool}$
  - $(\mathrm{Bool} \longrightarrow \mathrm{Bool}) \longrightarrow (\mathrm{Bool} \longrightarrow \mathrm{Bool})$

Typing rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-If)}$$

$$\lambda x{:}\,T_1.\, t_2 \;:\; T_1 \longrightarrow T_2 \qquad \text{(T-Abs)}$$

Typing rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{\texttt{t}_1 : \texttt{Bool} \quad \texttt{t}_2 : \texttt{T} \quad \texttt{t}_3 : \texttt{T}}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}} \qquad \text{(T-If)}$$

$$\frac{???}{\lambda x : T_1 . t_2 : T_1 \longrightarrow T_2} \qquad \text{(T-Abs)}$$

Typing context

- Γ is a sequence of variables and their types, and the "," operator extends Γ by adding *a new binding* on the right.
  - The empty context is sometimes written $\emptyset$; but usually we just omit it by writing $\vdash t : T$ for "The closed term t has type T under the empty set of assumptions."
- To avoid confusion between the new binding and any bindings that may already appear in Γ, we require that the name x be chosen so that it is distinct from the variables bound by Γ
  - variables bound by λ-abstractions may be renamed whenever convenient
- Γ can thus be thought of as *a finite function* from *variables* to their *types*.
  - Following this intuition, we write $dom(\Gamma)$ for the set of variables bound by Γ .

$\lambda_{\rightarrow}$

- What is the relation between these two statements?

  $$t : T$$
  $$\vdash t : T$$

  these two relations are *completely different things*.

  – for the *simple language of numbers* and *booleans*, typing is a *binary relation* between *terms* and *types* ($t : T$). We are dealing with *several different small programming languages*, each with *its own typing relation* (between *terms* in that language and *types* in that language)

  – for $\lambda_{\rightarrow}$, typing is a *ternary relation* between *contexts*, *terms*, and *types* ($\Gamma \vdash t : T$, $\vdash t : T$ if $\Gamma = \emptyset$)

$\lambda_{\rightarrow}$

Syntax

t	::=		*terms:*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
v	::=		*values:*
		λx:T.t	*abstraction value*
T	::=		*types:*
		T→T	*type of functions*
Γ	::=		*contexts:*
		∅	*empty context*
		Γ, x:T	*term variable binding*

Assume: all variables in Γ are different via renaming/internal

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \rightarrow T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

Type Derivation Tree

What derivations justify the following typing statement?

$$\vdash (\lambda x\!: \mathrm{Bool}.\ x)\ \mathrm{true} : \mathrm{Bool}$$

$$\dfrac{\dfrac{\dfrac{\mathtt{x{:}Bool} \in \mathtt{x{:}Bool}}{\mathtt{x{:}Bool} \vdash \mathtt{x} : \mathtt{Bool}}\ \text{T-VAR}}{\vdash \lambda\mathtt{x{:}Bool.x} : \mathtt{Bool}\to\mathtt{Bool}}\ \text{T-ABS} \qquad \dfrac{}{\vdash \mathtt{true} : \mathtt{Bool}}\ \text{T-TRUE}}{\vdash (\lambda\mathtt{x{:}Bool.x})\ \mathtt{true} : \mathtt{Bool}}\ \text{T-APP}$$

Properties of Typing

Inversion Lemma

Uniqueness of Types

Canonical Forms

Safety: Progress + Preservation

Inversion Lemma

1. If $\Gamma \vdash \texttt{true} : R$, then $R = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : R$, then $R = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : R$, then $\Gamma \vdash t_1 : \texttt{Bool}$ and $\Gamma \vdash t_2, t_3 : R$.
4. If $\Gamma \vdash x : R$, then $x{:}R \in \Gamma$.
5. If $\Gamma \vdash \lambda x{:}T_1 . t_2 : R$, then $R = T_1 \to R_2$ for some R_2 with $\Gamma, x{:}T_1 \vdash t_2 : R_2$.
6. If $\Gamma \vdash t_1 \; t_2 : R$, then there is some type T_{11} such that $\Gamma \vdash t_1 : T_{11} \to R$ and $\Gamma \vdash t_2 : T_{11}$.

Exercise: Is there any context Γ and type T such that $\Gamma \vdash x \; x : T$?

Canonical Forms

- Lemma:

1. If $\mathtt{v}$ is a value of type $\mathtt{Bool}$, then $\mathtt{v}$ is either $\mathtt{true}$ or $\mathtt{false}$.
2. If $\mathtt{v}$ is a value of type $\mathtt{T_1 \to T_2}$, then $\mathtt{v}$ has the form $\lambda\mathtt{x}:\mathtt{T_1}.\mathtt{t_2}$.

Uniqueness of Types

- **Theorem** [*Uniqueness of Types*]:

  In a ***given typing context*** *Γ*, a term *t* (with free variables all in the domain of *Γ*) has *at most one type*.

  Moreover, there is just *one derivation* of this typing built from the *inference rules* that generate the typing relation.

Progress

- **Theorem** [Progress]:

  Suppose t is a *closed*, *well-typed term,* then either t is a value or else there is some t′ with $t \rightarrow t'$.

  – *Closed*: No free variable

  – *Well-typed*: $\vdash t : T$ for some T

- Proof: same steps as before...

  – inversion lemma for typing relation

  – canonical forms lemma

  – progress theorem

Progress

- **Theorem** [Progress]:

  Suppose t is a *closed, well-typed term*. Then either t is a value or else there is some t′ with $t \rightarrow t'$.

  Proof: By induction on typing derivations.

  - The cases for *Boolean constants* and *conditions* are the same as before.
  - The *variable case* is trivial (cannot occur because t is closed).
  - The *abstraction case* is immediate, since abstractions are values.
  - The *case for application*, where $t = t_1\ t_2$ with $\vdash t_1 : T_{11} \rightarrow T_{12}$ and $\vdash t_2 : T_{11}$. By the induction hypothesis, either t_1 is a value or else it can make a step of evaluation, and likewise t_2.

  If t_1 can take a step, then rule E-App1 applies to t.

  If t_1 is a value and t_2 can take a step, then rule E-App2 applies.

  Finally, if both t_1 and t_2 are values, then the canonical forms lemma tells us that t_1 has the form $\lambda x{:}\ T_{11}.\ t_{12}$, and so rule E-AppAbs applies to t.

Preservation

- **Substitution Lemma** [Preservation of types under substitution]:

  if $\Gamma, x: S \vdash t: T$ and $\Gamma \vdash s: S$, then $\Gamma \vdash [x \mapsto s]\, t: T$.

  Proof: By induction on derivation of the statement $\Gamma, x: S \vdash t : T$

  proceed by *cases* on the possible *shape of t*.

- **Theorem** [Preservation]:

  If $\Gamma \vdash t: T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

  Proof: By induction on typing derivations.

  leaf as homework

The Curry-Howard Correspondence

- A connection between logic and type theory

Logic	Programming languages
propositions	types
proposition $P \supset Q$	type P→Q
proposition $P \land Q$	type P × Q (see §11.6)
proof of proposition P	term t of type P
proposition P is provable	type P is inhabited (by some term)

Erasure and Typability

- Types are used during *type checking*, but *do not need to appear* in the compiled form of the program.
- Terms in $\lambda_{\rightarrow}$ can be transformed to terms of *the untyped lambda-calculus* simply by *erasing type annotations* on lambda-abstractions.

$$
\begin{aligned}
\mathit{erase}(\mathtt{x}) &= \mathtt{x} \\
\mathit{erase}(\lambda\mathtt{x}:\mathtt{T_1}.\ \mathtt{t_2}) &= \lambda\mathtt{x}.\ \mathit{erase}(\mathtt{t_2}) \\
\mathit{erase}(\mathtt{t_1}\ \mathtt{t_2}) &= \mathit{erase}(\mathtt{t_1})\ \mathit{erase}(\mathtt{t_2})
\end{aligned}
$$

Erasure and Typability

- Conversely, an untyped λ-term m is said to be *typable* if there is some term t in the simply typed λ -calculus, some type T, and some context Γ such that

  $$erase(t) = m \text{ and } \Gamma \vdash t : T$$

  This process is called *type reconstruction* or *type inference*.

THEOREM:

1. If $\mathtt{t} \longrightarrow \mathtt{t}'$ under the typed evaluation relation, then $erase(\mathtt{t}) \longrightarrow erase(\mathtt{t}')$.
2. If $erase(\mathtt{t}) \longrightarrow \mathtt{m}'$ under the typed [untyped] evaluation relation, then there is a simply typed term $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$ and $erase(\mathtt{t}') = \mathtt{m}'$. □

Curry-Style vs. Church-Style

- Curry Style
  - Syntax → Semantics → Typing (*Semantics is prior to typing*)
  - Semantics is defined on *untyped terms*
  - Often used for *implicit* typed languages

- Church Style
  - Syntax → Typing → Semantics (*typing is prior to semantic*)
  - Semantics is defined only on *well-typed terms*
  - Often used for *explicit* typed languages

Homework

- Read through Chapter 9.
- Do Exercise 9.3.9.

THEOREM [PRESERVATION]: If $\Gamma \vdash \mathtt{t} : \mathsf{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathsf{T}$. □

Proof: EXERCISE [RECOMMENDED, ★★★]. The structure is very similar to the proof of the type preservation theorem for arithmetic expressions (8.3.3), except for the use of the substitution lemma. □