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 2026

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* , $\lambda_{\to}$ or *STLC*, for short.
- Talking about $\lambda_{\to}$, we always begin with *some set of "base types"*, unlike the *untyped lambda-calculus*, the "pure" form of $\lambda_{\to}$ (with no primitive values or operations) is not very interesting
  - Strictly speaking, there are ***many variants*** of $\lambda_{\to}$, 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
$T \longrightarrow T$	type of functions

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

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 ∅ ; but usually we just omit it by writing ⊢ 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(Γ) for the set of variables bound by Γ .

$\lambda_{\to}$

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

$\lambda_{\to}$

Syntax

t ::=		*terms:*
	x	*variable*
	$\lambda x:T.t$	*abstraction*
	t t	*application*
v ::=		*values:*
	$\lambda x:T.t$	*abstraction value*
T ::=		*types:*
	$T \to T$	*type of functions*
Γ ::=		*contexts:*
	$\varnothing$	*empty context*
	$\Gamma, x:T$	*term variable binding*

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

Evaluation $t \longrightarrow t'$

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

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

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

Typing $\Gamma \vdash t : T$

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

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

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

Type Derivation Tree

What derivations justify the following typing statement?

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

$$\dfrac{\dfrac{\dfrac{x:\text{Bool} \in x:\text{Bool}}{x:\text{Bool} \vdash x : \text{Bool}}\ \text{T-Var}}{\vdash \lambda x:\text{Bool}.x : \text{Bool}\to\text{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\vdash \texttt{true} : \text{Bool}}\ \text{T-True}}{\vdash (\lambda x:\text{Bool}.x)\ \texttt{true} : \text{Bool}}\ \text{T-App}$$

Properties of Typing

Inversion Lemma

Uniqueness of Types

Canonical Forms

Safety: Progress + Preservation

Inversion Lemma

1. If $\Gamma \vdash \texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\Gamma \vdash \texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : \texttt{R}$, then $\Gamma \vdash t_1 : \texttt{Bool}$ and $\Gamma \vdash t_2, t_3 : \texttt{R}$.
4. If $\Gamma \vdash \texttt{x} : \texttt{R}$, then $\texttt{x}:\texttt{R} \in \Gamma$.
5. If $\Gamma \vdash \lambda \texttt{x}:\texttt{T}_1 . t_2 : \texttt{R}$, then $\texttt{R} = \texttt{T}_1 \rightarrow \texttt{R}_2$ for some $\texttt{R}_2$ with $\Gamma, \texttt{x}:\texttt{T}_1 \vdash t_2 : \texttt{R}_2$.
6. If $\Gamma \vdash t_1 \ t_2 : \texttt{R}$, then there is some type $\texttt{T}_{11}$ such that $\Gamma \vdash t_1 : \texttt{T}_{11} \rightarrow \texttt{R}$ and $\Gamma \vdash t_2 : \texttt{T}_{11}$.

Exercise: Is there any context Γ and type T such that Γ ⊢ x x: T?

Canonical Forms

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

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 $t \longrightarrow t'$.

  - *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 $t \longrightarrow t'$.

  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 $\vdash t_1 : T_{11} \to T_{12}$ 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

- **Theorem** [Preservation]:

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

  Proof: By induction on typing derivations.

  leave as homework

- **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*.

Summary & Conclusion

- $\lambda_{\rightarrow}$ is a typed extension of λ-calculus with static typing
  - providing a type system for ensuring validity and preventing type errors
  - offering strong type safety guarantees through Progress and Preservation theorems on the cost of trading off the expressive power of Turing-completeness
  - serving as the "assembly language" for the semantics of modern functional programming languages and remaining a primary object of study in type theory, programming language theory, and proof assistants

The Curry-Howard Correspondence

- A connection between logic and type theory

LOGIC	PROGRAMMING LANGUAGES
propositions	types
proposition $P \supset Q$	type `P→Q`
proposition $P \wedge Q$	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 $\boldsymbol{\lambda}_{\rightarrow}$ can be transformed to terms of *the untyped lambda-calculus* simply by *erasing type annotations* on lambda-abstractions.

$$
\begin{array}{lcl}
\mathit{erase}(\mathtt{x}) & = & \mathtt{x} \\
\mathit{erase}(\lambda \mathtt{x}{:}\mathtt{T}_1.\ \mathtt{t}_2) & = & \lambda \mathtt{x}.\ \mathit{erase}(\mathtt{t}_2) \\
\mathit{erase}(\mathtt{t}_1\ \mathtt{t}_2) & = & \mathit{erase}(\mathtt{t}_1)\ \mathit{erase}(\mathtt{t}_2)
\end{array}
$$

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 and Γ ⊢ t: T

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

THEOREM:

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

untyped

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 9.3.10.

9.3.10 EXERCISE [RECOMMENDED, ⋆⋆]: In Exercise 8.3.6 we investigated the *subject expansion* property for our simple calculus of typed arithmetic expressions. Does it hold for the "functional part" of the simply typed lambda-calculus? That is, suppose $\mathtt{t}$ does not contain any conditional expressions. Do $\mathtt{t} \longrightarrow \mathtt{t}'$ and $\Gamma \vdash \mathtt{t}' : \mathtt{T}$ imply $\Gamma \vdash \mathtt{t} : \mathtt{T}$? □