

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 2024





The Typing Relation t:T





- Values have two possible "shapes":
 - either *booleans*
 - or *numbers*.

T ::= Bool Nat *types type of booleans type of numbers*

Typing Rules



true : Bool	(T-TRUE)
false : Bool	(T-FALSE)
$\begin{array}{ccc} t_1: \texttt{Bool} & t_2: \texttt{T} & t_3: \texttt{T} \\ \\ \texttt{if} \ \texttt{t}_1 \ \texttt{then} \ \texttt{t}_2 \ \texttt{else} \ \texttt{t}_3: \texttt{T} \end{array}$	(T-IF)
0 : Nat	(T-Zero)
t_1 : Nat succ t_1 : Nat	(T-Succ)
$ frac{ t t_1 : extsf{Nat}}{ extsf{pred } t_1 : extsf{Nat}}$	(T-Pred)
$ t_1: Nat$ iszero t $_1: Bool$	(T-IsZero)

Design Principles of Programming Language, Spring 2024



• Definition:

the *typing relation* for arithmetic expressions is the *smallest binary relation* between *terms* and *types* satisfying **all instances** of the typing rules.

• A term *t* is *typable* (or *well typed*) if there is some *T* such that *t* : *T*.



Chapter 9: Simply Typed Lambda-Calculus

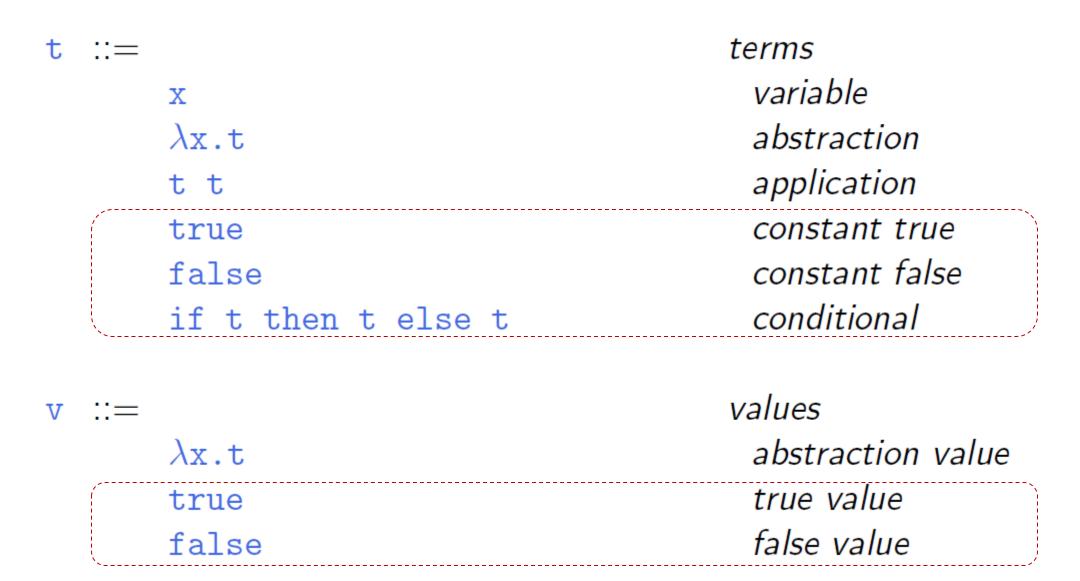
Function Types The Typing Relation Properties of Typing The Curry-Howard Correspondence Erasure and Typability

The simply typed lambda-calculus



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





Design Principles of Programming Language, Spring 2024

Function Types



- $T_1 \longrightarrow T_2$
 - *classifying functions* that expect arguments of type T1 and return results of type T2.
- the type constructor \rightarrow is right-associative, e.g.,

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

• Let's consider *Booleans* with lambda calculus

 $T ::= Bool \\ T \rightarrow T$

types : type of booleans type of functions

- Examples
 - Bool \rightarrow Bool
 - $(\mathsf{Bool} \to \mathsf{Bool}) \to (\mathsf{Bool} \to \mathsf{Bool})$

Typing rules



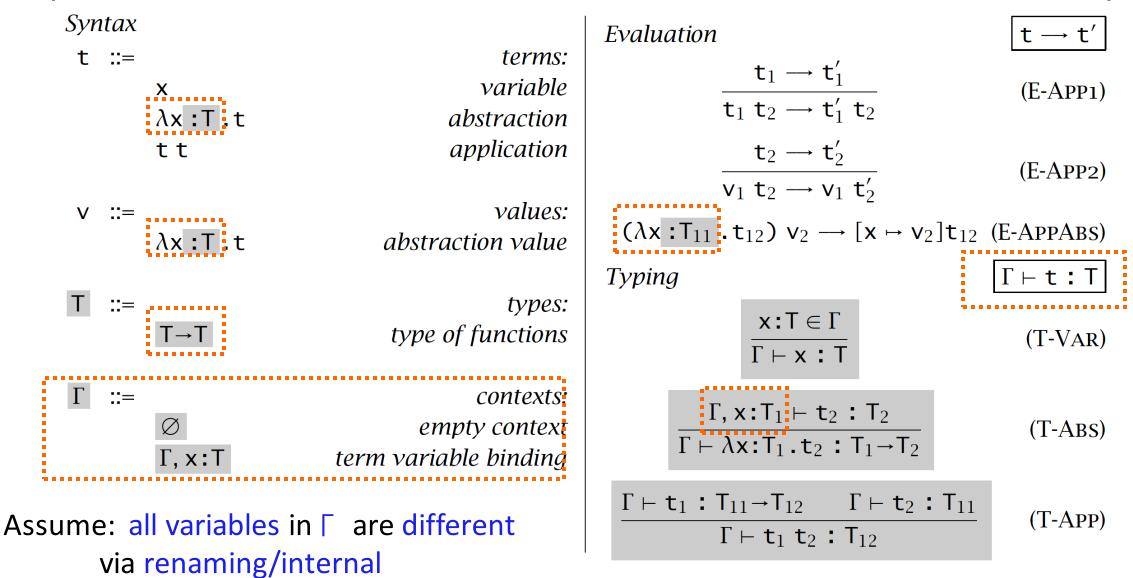
(T-TRUE)		e : Bool	true	
(T-FALSE)		e : Bool	fals	
(T-IF)	$t_3:T$	$t_2:T$	Bool	$t_1:$
	$t_3:T$	t_2 else	t_1 then	if

$$\frac{???}{\lambda x: T_1 \ t_2: \ T_1 \longrightarrow T_2}$$

 $(T-A_{BS})$

 λ_{\rightarrow}





Design Principles of Programming Language, Spring 2024





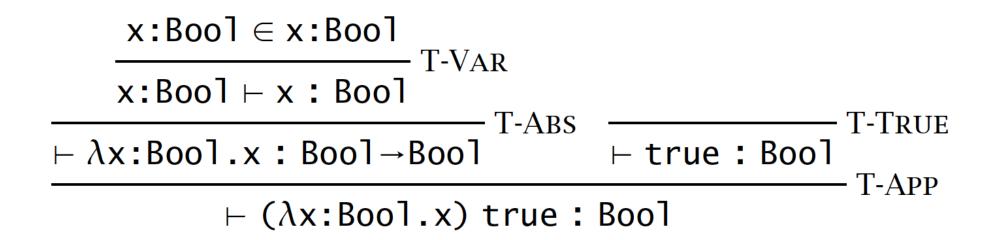
What is the relation between these two statements?
 t : T
 ⊢ t : T

these two relations are *completely different things*.

- 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 the simple language of numbers and booleans, typing is a binary relation between terms and types (t : T).
 - for λ_{\rightarrow} , typing is a *ternary relation* between *contexts*, *terms*, and *types* ($\Gamma \vdash t : T$, $\vdash t : T$ if $\Gamma = \emptyset$)



What derivations justify the following typing statement? $\vdash (\lambda x: Bool. x) true : Bool$





Properties of Typing

Inversion Lemma Uniqueness of Types Canonical Forms Safety: Progress + Preservation

Inversion Lemma



- 1. If $\Gamma \vdash \text{true}$: R, then R = Bool.
- 2. If $\Gamma \vdash false : R$, then R = Bool.
- 3. If $\Gamma \vdash if t_1$ then t_2 else $t_3 : R$, then $\Gamma \vdash t_1 :$ 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 $R = T_1 \rightarrow R_2$ 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 $\Gamma \vdash t_1 : T_{11} \rightarrow R$ and $\Gamma \vdash t_2 : T_{11}$.

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

Design Principles of Programming Language, Spring 2024



• Lemma:

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



• **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.



• 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: ⊢ 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 λx : T_{11} . t_{12} , and so rule E-AppAbs applies to t.



• Theorem [Preservation]:

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

Proof: By induction on typing derivations.

Substitution Lemma [Preservation of types under substitution]:
if Γ, x: S ⊢ t: T and Γ ⊢ s: S, then Γ ⊢ [x ↦ s] t: T. *Proof*: By induction on derivation of Γ, x: S ⊢ t : T *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.

Substitution Lemma [Preservation of types under substitution]:
if Γ, x: S ⊢ t: T and Γ ⊢ s: S, then Γ ⊢ [x ↦ s] t: T. *Proof*: By induction on derivation of Γ, x: S ⊢ t : T *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.

Substitution Lemma [Preservation of types under substitution]:
if Γ, x: S ⊢ t: T and Γ ⊢ s: S, then Γ ⊢ [x ↦ s] t: T. *Proof*: By induction on derivation of Γ, x: S ⊢ t : T *cases* on the possible *shape of t*.



• A connection between logic and type theory

Logic	PROGRAMMING LANGUAGES
propositions	types
proposition $P \supset Q$	type P→Q
proposition $P \wedge Q$	type $P \times Q$ (see §11.6)
proof of proposition P	term t of type P
proposition <i>P</i> is provable	type P is inhabited (by some term)



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

$$erase(x) = x$$

$$erase(\lambda x:T_1. t_2) = \lambda x. erase(t_2)$$

$$erase(t_1 t_2) = erase(t_1) erase(t_2)$$



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 \Gamma \vdash t: T
```

This process is called type reconstruction or type inference.

THEOREM:

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

Design Principles of Programming Language, Spring 2024

untyped

UNITE TO A

- Curry Style
 - Syntax \rightarrow Semantics \rightarrow Typing
 - Semantics is defined on untyped terms
 - Often used for *implicit* typed languages
- Church Style
 - Syntax \rightarrow Typing \rightarrow Semantics
 - 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 $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof: EXERCISE [RECOMMENDED, $\star \star \star$]. 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.