Lecture 7: Simple Types and Simply-Typed Lambda Calculus

Prof. Liang Huang
huang@qc.cs.cuny.edu
Types

\[
t ::= \\
  \text{true} \\
  \text{false} \\
  \text{if } t \text{ then } t \text{ else } t \\
  \text{pair } t \ t \\
  \text{fst } t \\
  \text{snd } t \\
\]

\[
v ::= \\
  \text{true} \\
  \text{false} \\
  \text{pair } v \ v \\
\]

stuck terms?

how to fix it?

\[
\begin{align*}
  S & \rightarrow \text{NP VP} \\
  S & \rightarrow S \ \text{conj} \ S \\
  \text{NP} & \rightarrow \text{Noun} \\
  \text{NP} & \rightarrow \text{Det Noun} \\
  \text{NP} & \rightarrow \text{NP PP} \\
  \text{NP} & \rightarrow \text{NP conj NP} \\
  \text{VP} & \rightarrow \text{Verb} \\
  \text{VP} & \rightarrow \text{Verb NP} \\
  \text{VP} & \rightarrow \text{Verb NP NP} \\
  \text{VP} & \rightarrow \text{VP PP} \\
  \text{PP} & \rightarrow \text{P NP}
\end{align*}
\]
Plan

- **First**, we’ll go back to the simple language of arithmetic and boolean expressions and show how to equip it with a (very simple) type system.
- The key property of this type system will be *soundness*: Well-typed programs do not get stuck.
- **Next**, we’ll develop a simple type system for the lambda-calculus.
- We’ll spend a good part of the rest of the semester adding features to this type system.
Outline

1. begin with a set of terms, a set of values, and an evaluation relation
2. define a set of *types* classifying values according to their “shapes”
3. define a *typing relation* $t : T$ that classifies terms according to the shape of the values that result from evaluating them
4. check that the typing relation is *sound* in the sense that,
   4.1 if $t : T$ and $t \rightarrow^* v$, then $v : T$
   4.2 if $t : T$, then evaluation of $t$ will not get stuck
Review: Arithmetic Expressions – Syntax

\[ t \ ::= \]
\[ \text{true} \]
\[ \text{false} \]
\[ \text{if } t \text{ then } t \text{ else } t \]
\[ 0 \]
\[ \text{succ } t \]
\[ \text{pred } t \]
\[ \text{iszero } t \]

\[ v \ ::= \]
\[ \text{true} \]
\[ \text{false} \]
\[ \text{nv} \]

\[ \text{nv} \ ::= \]
\[ 0 \]
\[ \text{succ } \text{nv} \]
Evaluation Rules

\[
\begin{align*}
\text{if true then } t_2 \text{ else } t_3 & \rightarrow t_2 \quad (E-\text{IfTrue}) \\
\text{if false then } t_2 \text{ else } t_3 & \rightarrow t_3 \quad (E-\text{IfFalse})
\end{align*}
\]

\[
\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \quad (E-\text{If})
\]
\[
\begin{align*}
&\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'} \quad \text{(E-Succ)} \\
&\frac{\text{pred } 0 \rightarrow 0}{\text{(E-PredZero)}} \\
&\frac{\text{pred } (\text{succ } n_{v_1}) \rightarrow n_{v_1}}{\text{(E-PredSucc)}} \\
&\frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'} \quad \text{(E-Pred)} \\
&\frac{\text{iszero } 0 \rightarrow \text{true}}{\text{(E-IszeroZero)}} \\
&\frac{\text{iszero } (\text{succ } n_{v_1}) \rightarrow \text{false}}{\text{(E-IszeroSucc)}} \\
&\frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'} \quad \text{(E-Iszero)}}
\end{align*}
\]
Types

In this language, values have two possible “shapes”: they are either booleans or numbers.

\[ T ::= \]

- \texttt{Bool} \quad \text{types}
- \texttt{Nat} \quad \text{type of booleans}
- \text{type of numbers}
Typing Rules

true : Bool  \quad \text{(T-TRUE)}
false : Bool \quad \text{(T-FALSE)}

\[
\begin{array}{c}
t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T \\
\frac{}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}
\end{array}
\quad \text{(T-IF)}
\]

0 : Nat \quad \text{(T-ZERO)}

\[
\begin{array}{c}
t_1 : \text{Nat} \\
\frac{}{\text{succ } t_1 : \text{Nat}}
\end{array}
\quad \text{(T-SUCC)}
\]

\[
\begin{array}{c}
t_1 : \text{Nat} \\
\frac{}{\text{pred } t_1 : \text{Nat}}
\end{array}
\quad \text{(T-PRED)}
\]

\[
\begin{array}{c}
t_1 : \text{Nat} \\
\frac{}{\text{iszero } t_1 : \text{Bool}}
\end{array}
\quad \text{(T-ISZERO)}
\]
Typing Derivations

Every pair \((t, T)\) in the typing relation can be justified by a derivation tree built from instances of the inference rules.

Proofs of properties about the typing relation often proceed by induction on typing derivations.
Imprecision of Typing

Like other static program analyses, type systems are generally *imprecise*: they do not predict exactly what kind of value will be returned by every program, but just a conservative (safe) approximation.

\[
\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad (T\text{-IF})
\]

Using this rule, we cannot assign a type to

\[
\text{if true then 0 else false}
\]

even though this term will certainly evaluate to a number.
Properties of the Typing Relation
Type Safety

The safety (or soundness) of this type system can be expressed by two properties:

1. **Progress**: A well-typed term is not stuck
   
   If \( t : T \), then either \( t \) is a value or else \( t \rightarrow t' \) for some \( t' \).

2. **Preservation**: Types are preserved by one-step evaluation
   
   If \( t : T \) and \( t \rightarrow t' \), then \( t' : T \).
Inversion

Lemma:

1. If true : R, then \( R = \text{Bool} \).
2. If false : R, then \( R = \text{Bool} \).
3. If if \( t_1 \) then \( t_2 \) else \( t_3 \) : R, then \( t_1 : \text{Bool}, t_2 : R, \) and \( t_3 : R \).
4. If 0 : R, then \( R = \text{Nat} \).
5. If succ \( t_1 \) : R, then \( R = \text{Nat} \) and \( t_1 : \text{Nat} \).
6. If pred \( t_1 \) : R, then \( R = \text{Nat} \) and \( t_1 : \text{Nat} \).
7. If iszero \( t_1 \) : R, then \( R = \text{Bool} \) and \( t_1 : \text{Nat} \).

Proof:

This leads directly to a recursive algorithm for calculating the type of a term...
Inversion

Lemma:

1. If \( \text{true} : R \), then \( R = \text{Bool} \).
2. If \( \text{false} : R \), then \( R = \text{Bool} \).
3. If \( \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R \), then \( t_1 : \text{Bool} \), \( t_2 : R \), and \( t_3 : R \).
4. If \( 0 : R \), then \( R = \text{Nat} \).
5. If \( \text{succ } t_1 : R \), then \( R = \text{Nat} \) and \( t_1 : \text{Nat} \).
6. If \( \text{pred } t_1 : R \), then \( R = \text{Nat} \) and \( t_1 : \text{Nat} \).
7. If \( \text{iszero } t_1 : R \), then \( R = \text{Bool} \) and \( t_1 : \text{Nat} \).

Proof: \( \ldots \)
Inversion

Lemma:

1. If true : R, then R = Bool.
2. If false : R, then R = Bool.
3. If if t₁ then t₂ else t₃ : R, then t₁ : Bool, t₂ : R, and t₃ : R.
4. If 0 : R, then R = Nat.
5. If succ t₁ : R, then R = Nat and t₁ : Nat.
6. If pred t₁ : R, then R = Nat and t₁ : Nat.
7. If iszero t₁ : R, then R = Bool and t₁ : Nat.

Proof: ...
Typechecking Algorithm

typeof(t) = if t = true then Bool
    else if t = false then Bool
    else if t = if t1 then t2 else t3 then
        let T1 = typeof(t1) in
        let T2 = typeof(t2) in
        let T3 = typeof(t3) in
        if T1 = Bool and T2=T3 then T2
        else "not typable"
    else if t = 0 then Nat
    else if t = succ t1 then
        let T1 = typeof(t1) in
        if T1 = Nat then Nat else "not typable"
    else if t = pred t1 then
        let T1 = typeof(t1) in
        if T1 = Nat then Nat else "not typable"
    else if t = iszero t1 then
        let T1 = typeof(t1) in
        if T1 = Nat then Bool else "not typable"
Canonical Forms

Lemma:

1. If $v$ is a value of type $\text{Bool}$, then $v$ is either $\text{true}$ or $\text{false}$.
2. If $v$ is a value of type $\text{Nat}$, then $v$ is a numeric value.

Proof:
Lemma:

1. If $v$ is a value of type $\text{Bool}$, then $v$ is either $\text{true}$ or $\text{false}$.
2. If $v$ is a value of type $\text{Nat}$, then $v$ is a numeric value.

Proof: Recall the syntax of values:

$v ::= $

$\text{true}$
$\text{false}$
$\text{nv}$

$\text{nv} ::= $

$0$
$succ \text{ nv}$

For part 1,
Lemma:

1. If \( v \) is a value of type \( \text{Bool} \), then \( v \) is either \( \text{true} \) or \( \text{false} \).
2. If \( v \) is a value of type \( \text{Nat} \), then \( v \) is a numeric value.

Proof: Recall the syntax of values:

\[
\begin{align*}
  v & ::= \quad \text{values} \\
    & \quad \text{true value} \\
    & \quad \text{false value} \\
    & \quad \text{numeric value} \\
  \text{nv} & ::= \quad \text{numeric values} \\
    & \quad \text{zero value} \\
    & \quad \text{successor value}
\end{align*}
\]

For part 1, if \( v \) is \( \text{true} \) or \( \text{false} \), the result is immediate.
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 \texttt{Nat}, then \( v \) is a numeric value.

Proof: Recall the syntax of values:

\[
\begin{align*}
\texttt{v} & ::= \quad \text{values} \\
\quad \texttt{true} & \quad \text{true value} \\
\quad \texttt{false} & \quad \text{false value} \\
\quad \texttt{nv} & \quad \text{numeric value} \\
\texttt{nv} & ::= \quad \text{numeric values} \\
\quad \texttt{0} & \quad \text{zero value} \\
\quad \texttt{succ \ nv} & \quad \text{successor value}
\end{align*}
\]

For part 1, if \( v \) is \texttt{true} or \texttt{false}, the result is immediate. But \( v \) cannot be \texttt{0} or \texttt{succ \ nv}, since the inversion lemma tells us that \( v \) would then have type \texttt{Nat}, not \texttt{Bool}.
Lemma:

1. If $v$ is a value of type $\text{Bool}$, then $v$ is either $\text{true}$ or $\text{false}$.
2. If $v$ is a value of type $\text{Nat}$, then $v$ is a numeric value.

Proof: Recall the syntax of values:

$$
v ::= \begin{array}{l}
\text{true} \quad \text{true value} \\
\text{false} \quad \text{false value} \\
\text{nv} \quad \text{numeric value} \\
\text{nv} ::= \begin{array}{l}
0 \quad \text{zero value} \\
\text{succ} \text{ nv} \quad \text{successor value}
\end{array}
\end{array}
$$

For part 1, if $v$ is $\text{true}$ or $\text{false}$, the result is immediate. But $v$ cannot be $0$ or $\text{succ} \text{ nv}$, since the inversion lemma tells us that $v$ would then have type $\text{Nat}$, not $\text{Bool}$. Part 2 is similar.
Progress

Theorem: Suppose \( t \) is a well-typed term (that is, \( t : T \) for some \( T \)). Then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).
Progress

Theorem: Suppose \( t \) is a well-typed term (that is, \( t : T \) for some \( T \)). Then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).

Proof:
Progress

Theorem: Suppose \( t \) is a well-typed term (that is, \( t : T \) for some \( T \)). Then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).

Proof: By induction on a derivation of \( t : T \).
**Progress**

*Theorem:* Suppose $t$ is a well-typed term (that is, $t : T$ for some $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

*Proof:* By induction on a derivation of $t : T$.

The $T$-True, $T$-False, and $T$-Zero cases are immediate, since $t$ in these cases is a value.
**Theorem:** Suppose \( t \) is a well-typed term (that is, \( t : T \) for some \( T \)). Then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).

**Proof:** By induction on a derivation of \( t : T \).

The **T-True**, **T-False**, and **T-Zero** cases are immediate, since \( t \) in these cases is a value.

**Case T-If:**
\[
\begin{align*}
  t &= \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\
  t_1 &: \text{Bool} \quad t_2 &: T \quad t_3 &: T
\end{align*}
\]
**Theorem:** Suppose $t$ is a well-typed term (that is, $t : T$ for some $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

**Proof:** By induction on a derivation of $t : T$.

The $T\text{-True}$, $T\text{-False}$, and $T\text{-Zero}$ cases are immediate, since $t$ in these cases is a value.

**Case $T\text{-If}$:**

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

$t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

By the induction hypothesis, either $t_1$ is a value or else there is some $t'_1$ such that $t_1 \rightarrow t'_1$. If $t_1$ is a value, then the canonical forms lemma tells us that it must be either $\text{true}$ or $\text{false}$, in which case either $E\text{-IfTrue}$ or $E\text{-IfFalse}$ applies to $t$. On the other hand, if $t_1 \rightarrow t'_1$, then, by $E\text{-If}$, $t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$. 
Preservation

*Theorem:* If $t : T$ and $t \rightarrow t'$, then $t' : T$. 
Preservation

Theorem: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

Proof: By induction on the given typing derivation.
Preservation

*Theorem:* If $t : T$ and $t \rightarrow t'$, then $t' : T$.

*Proof:* By induction on the given typing derivation.

*Case T-True:* $t = \text{true}$ $T = \text{Bool}$
Then $t$ is a value, so it cannot be that $t \rightarrow t'$ for any $t'$, and the theorem is vacuously true.
Preservation

Theorem: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

Proof: By induction on the given typing derivation.

Case T-If:

$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \quad t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

There are three evaluation rules by which $t \rightarrow t'$ can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.
Preservation

*Theorem:* If $t : T$ and $t \rightarrow t'$, then $t' : T$.

*Proof:* By induction on the given typing derivation.

**Case T-If:**

\[ t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \quad t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T \]

There are three evaluation rules by which $t \rightarrow t'$ can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.

**Subcase E-IfTrue:**

\[ t_1 = \text{true} \quad t' = t_2 \]

Immediate, by the assumption $t_2 : T$.

(E-IfFalse subcase: Similar.)
**Preservation**

*Theorem:* If \( t : T \) and \( t \rightarrow t' \), then \( t' : T \).

*Proof:* By induction on the given typing derivation.

*Case* T-If: 
\[
t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \quad t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T
\]

There are three evaluation rules by which \( t \rightarrow t' \) can be derived: E-IfTrue, E-IfFalse, and E-If. Consider each case separately.

*Subcase* E-If: 
\[
t_1 \rightarrow t_1' \quad t' = \text{if } t_1' \text{ then } t_2 \text{ else } t_3
\]
Applying the IH to the subderivation of \( t_1 : \text{Bool} \) yields \( t_1' : \text{Bool} \). Combining this with the assumptions that \( t_2 : T \) and \( t_3 : T \), we can apply rule T-If to conclude that \( \text{if } t_1' \text{ then } t_2 \text{ else } t_3 : T \), that is, \( t' : T \).
Recap: Type Systems

- Very successful example of a *lightweight formal method*
- big topic in PL research
- enabling technology for all sorts of other things, e.g. language-based security
- the skeleton around which modern programming languages are designed
The Simply Typed Lambda-Calculus
The simply typed lambda-calculus

The system we are about to define is commonly called the simply typed lambda-calculus, or \( \lambda \rightarrow \) for short.

Unlike the untyped lambda-calculus, the “pure” form of \( \lambda \rightarrow \) (with no primitive values or operations) is not very interesting; to talk about \( \lambda \rightarrow \), we always begin with some set of “base types.”

- So, 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
“Simple Types”

\[ T ::= \]
\[ \text{Bool} \]
\[ T \to T \]

*types*

*type of booleans*

*types of functions*
We now have a choice to make. Do we...

- annotate lambda-abstractions with the expected type of the argument
  \[ \lambda x : T_1 . \ t_2 \]
  (as in most mainstream programming languages), or
- continue to write lambda-abstractions as before
  \[ \lambda x . \ t_2 \]
  and ask the typing rules to “guess” an appropriate annotation
  (as in OCaml)?

Both are reasonable choices, but the first makes the job of defining the typing rules simpler. Let’s take this choice for now.
Typing rules

true : Bool

false : Bool

\[
\begin{align*}
t_1 : \text{Bool} & \quad t_2 : T & \quad t_3 : T \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T
\end{align*}
\]

(T-True)

(T-False)

(T-If)
Typing rules

- **true**: \( \text{Bool} \)  
  \( (T-\text{TRUE}) \)

- **false**: \( \text{Bool} \)  
  \( (T-\text{FALSE}) \)

- **if**: \( t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T \)  
  \[ \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \]  
  \( (T-\text{IF}) \)

- **\( \lambda x : T_1. t_2 : T_1 \rightarrow T_2 \)**  
  \( (T-\text{ABS}) \)
Typing rules

true : Bool

false : Bool

\( \frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \)  \( \text{(T-IF)} \)

\( \Gamma, x : T_1 \vdash t_2 : T_2 \)

\( \frac{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2}{\Gamma \vdash \lambda x : T. t_1 t_2 : T} \)  \( \text{(T-ABS)} \)

\( x : T \in \Gamma \)

\( \frac{\Gamma \vdash x : T}{\Gamma \vdash x : T} \)  \( \text{(T-VAR)} \)
Typing rules

\[ \Gamma \vdash \text{true} : \text{Bool} \]  \hspace{1cm} (T-TRUE)

\[ \Gamma \vdash \text{false} : \text{Bool} \]  \hspace{1cm} (T-FALSE)

\[ \Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T \]
\[ \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \]  \hspace{1cm} (T-IF)

\[ \Gamma, x : T_1 \vdash t_2 : T_2 \]
\[ \Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \]  \hspace{1cm} (T-ABS)

\[ x : T \in \Gamma \]
\[ \Gamma \vdash x : T \]  \hspace{1cm} (T-VAR)

\[ \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \]
\[ \Gamma \vdash t_1 \; t_2 : T_{12} \]  \hspace{1cm} (T-APP)
Typing Derivations

What derivations justify the following typing statements?

1. \( \vdash (\lambda x : \text{Bool}. x) \text{true} : \text{Bool} \)
2. \( f : \text{Bool} \rightarrow \text{Bool} \vdash f \ (\text{if false then true else false}) : \text{Bool} \)
3. \( f : \text{Bool} \rightarrow \text{Bool} \vdash \lambda x : \text{Bool}. f \ (\text{if x then false else x}) : \text{Bool} \rightarrow \text{Bool} \)
Typed based on $\lambda$ (5-3)

**Syntax**

$t ::= x \quad \text{variable}
\lambda x : T . t \quad \text{abstraction}
\app t t \quad \text{application}$

$v ::= \lambda x : T . t \quad \text{abstraction value}$

$T ::= T \rightarrow T \quad \text{type of functions}$

$\Gamma ::= \emptyset \quad \text{empty context}
\Gamma, x : T \quad \text{term variable binding}$

**Evaluation**

$$
\begin{align*}
\frac{t_1 \to t'_1}{t_1 \app t_2 \to t'_1 t_2} \\
\frac{t_2 \to t'_2}{v_1 \app t_2 \to v_1 t'_2} \\
(\lambda x : T_{11} . t_{12}) v_2 \to [x \mapsto v_2] t_{12}
\end{align*}
$$

(E-APP1)  
(E-APP2)  
(E-APPABS)

**Typing**

$$
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}
\frac{\Gamma \vdash \lambda x : T_{1} . t_{2} : T_{1} \rightarrow T_{2}}{\Gamma \vdash \lambda x : T_{11} \rightarrow T_{12} . t_{2} : T_{11} \rightarrow T_{12}} \quad \text{(T-APP)}
\frac{\Gamma \vdash \lambda x : T_{11} . t_{12} : T_{11} \rightarrow T_{12}}{\Gamma \vdash t_{1} \app t_{2} : T_{12}} \quad \text{(T-ABS)}
$$
Properties of $\lambda\to$

The fundamental property of the type system we have just defined is *soundness* with respect to the operational semantics.

1. **Progress:** A closed, well-typed term is not stuck

   $\text{If } \Gamma \vdash t : T, \text{ then either } t \text{ is a value or else } t \rightarrow t'$

   for some $t'$.

2. **Preservation:** Types are preserved by one-step evaluation

   $\text{If } \Gamma \vdash t : T \text{ and } t \rightarrow t', \text{ then } \Gamma \vdash t' : T.$
Proving progress

Same steps as before...
Proving progress

Same steps as before...

- inversion lemma for typing relation
- canonical forms lemma
- progress theorem
Inversion

Lemma:

1. If $\Gamma \vdash \text{true} : R$, then $R = \text{Bool}$.
2. If $\Gamma \vdash \text{false} : R$, then $R = \text{Bool}$.
3. If $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $\Gamma \vdash t_1 : \text{Bool}$ and $\Gamma \vdash t_2, t_3 : R$. 
Inversion

Lemma:

1. If $\Gamma \vdash \text{true} : R$, then $R = \text{Bool}$.
2. If $\Gamma \vdash \text{false} : R$, then $R = \text{Bool}$.
3. If $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $\Gamma \vdash t_1 : \text{Bool}$ and $\Gamma \vdash t_2, t_3 : R$.
4. If $\Gamma \vdash x : R$, then
Inversion

Lemma:

1. If $\Gamma \vdash \text{true} : R$, then $R = \text{Bool}$.
2. If $\Gamma \vdash \text{false} : R$, then $R = \text{Bool}$.
3. If $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $\Gamma \vdash t_1 : \text{Bool}$ and $\Gamma \vdash t_2, t_3 : R$.
4. If $\Gamma \vdash x : R$, then $x : R \in \Gamma$. 
Lemma:

1. If $\Gamma \vdash \text{true} : R$, then $R = \text{Bool}$.
2. If $\Gamma \vdash \text{false} : R$, then $R = \text{Bool}$.
3. If $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $\Gamma \vdash t_1 : \text{Bool}$ 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
Inversion

Lemma:

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

Lemma:

1. If $\Gamma \vdash \text{true} : R$, then $R = \text{Bool}$.
2. If $\Gamma \vdash \text{false} : R$, then $R = \text{Bool}$.
3. If $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $\Gamma \vdash t_1 : \text{Bool}$ 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
Inversion

Lemma:

1. If $\Gamma \vdash \text{true} : R$, then $R = \text{Bool}$.
2. If $\Gamma \vdash \text{false} : R$, then $R = \text{Bool}$.
3. If $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R$, then $\Gamma \vdash t_1 : \text{Bool}$ 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 $\Gamma \vdash t_2 : T_{11}$.
Canonical Forms

Lemma:
Canonical Forms

Lemma:

1. If $v$ is a value of type $\text{Bool}$, then
Canonical Forms

Lemma:

1. If \( v \) is a value of type \( \text{Bool} \), then \( v \) is either \text{true} or \text{false}. 
Canonical Forms

Lemma:

1. If $v$ is a value of type $\text{Bool}$, then $v$ is either $\text{true}$ or $\text{false}$.
2. If $v$ is a value of type $T_1 \rightarrow T_2$, then
Lemma:

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

*Theorem:* Suppose $t$ is a closed, well-typed term (that is, $\vdash t : T$ for some $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

*Proof:* By induction
Progress

*Theorem:* Suppose $t$ is a closed, well-typed term (that is, $\vdash t : T$ for some $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

*Proof:* By induction on typing derivations.
**Theorem:** Suppose \( t \) is a closed, well-typed term (that is, \( \vdash t : T \) for some \( T \)). 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 (because \( t \) is closed). The abstraction case is immediate, since abstractions are values.
Progress

Theorem: Suppose \( t \) is a closed, well-typed term (that is, \( \vdash t : T \) for some \( T \)). 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 (because \( t \) is closed). The abstraction case is immediate, since abstractions are values.
Consider the case for application, where \( t = t_1 \ t_2 \) with \( \vdash t_1 : T_{11} \rightarrow T_{12} \) and \( \vdash t_2 : T_{11} \).
Progress

*Theorem:* Suppose $t$ is a closed, well-typed term (that is, $\vdash t : T$ for some $T$). 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 (because $t$ is closed). The abstraction case is immediate, since abstractions are values.

Consider 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 $t_2$. 

**Progress**

*Theorem:* Suppose $t$ is a closed, well-typed term (that is, $\vdash t : T$ for some $T$). 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 (because $t$ is closed). The abstraction case is immediate, since abstractions are values.

Consider 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

Theorem: If \( \Gamma \vdash t : T \) and \( t \to t' \), then \( \Gamma \vdash t' : T \).

Proof: By induction
**Preservation**

*Theorem:* If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

*Proof:* By induction on typing derivations.

Which case is the hard one??
Preservation

**Theorem:** If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

**Proof:** By induction on typing derivations.

**Case T-App:** Given $t = t_1 \ t_2$

- $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$
- $\Gamma \vdash t_2 : T_{11}$
- $T = T_{12}$

Show $\Gamma \vdash t' : T_{12}$
Preservation

*Theorem:* If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

*Proof:* By induction on typing derivations.

Case $T$-App: Given $t = t_1 \ t_2$

- $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$
- $\Gamma \vdash t_2 : T_{11}$
- $T = T_{12}$

Show $\Gamma \vdash t' : T_{12}$

By the inversion lemma for evaluation, there are three subcases...
Preservation

**Theorem:** If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

**Proof:** By induction on typing derivations.

**Case T-App:** Given $t = t_1 \ t_2$

1. $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$
2. $\Gamma \vdash t_2 : T_{11}$
3. $T = T_{12}$

Show $\Gamma \vdash t' : T_{12}$

By the inversion lemma for evaluation, there are three subcases...

**Subcase:** $t_1 = \lambda x : T_{11} \ . \ t_{12}$
1. $t_2$ a value $v_2$
2. $t' = [x \mapsto v_2] t_{12}$
Preservation

**Theorem:** If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

**Proof:** By induction on typing derivations.

**Case $T$-App:** Given $t = t_1 t_2$

$\Gamma \vdash t_1 : T_{11} \rightarrow T_{12}$

$\Gamma \vdash t_2 : T_{11}$

$T = T_{12}$

Show $\Gamma \vdash t' : T_{12}$

By the inversion lemma for evaluation, there are three subcases...

**Subcase:** $t_1 = \lambda x : T_{11}. t_{12}$

$t_2$ a value $v_2$

$t' = [x \mapsto v_2]t_{12}$

Uh oh.
The “Substitution Lemma”

*Lemma:* Types are preserved under substitution.

That is, if $\Gamma, \; x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$. 
The “Substitution Lemma”

Lemma: Types are preserved under substitution.
That is, if $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.

Proof: ...
Preservation

Recommended: Complete the proof of preservation