This Lecture

- Recapitulation: our example language, stuck terms, type systems.
- Basic typing rules
- Safety = Progress + Preservation
- Extensions: typing let-expressions and functions

Much of this lecture follows parts of the first few chapters of B. C. Pierce 2002 *Types and Programming Languages* closely.
Recap: Example Language

Abstract syntax for the example language:

\[ t \rightarrow \]

- **true**
- **false**
- **if** \( t \) **then** \( t \) **else** \( t \)
- **0**
- **succ** \( t \)
- **pred** \( t \)
- **iszero** \( t \)

**terms:**

- constant true
- constant false
- conditional
- constant zero
- successor
- predecessor
- zero test
Recap: Values

The *values* of a language are a subset of the terms that are *possible results of evaluation*.

\[
\begin{align*}
\nu \rightarrow & \quad \text{values:} \\
\quad & \text{true} \quad \text{true value} \\
\quad & \text{false} \quad \text{false value} \\
\quad & \nu \quad \text{numeric value} \\
\nu \rightarrow & \quad \text{numeric values:} \\
\quad & 0 \quad \text{zero value} \\
\quad & \text{succ } \nu \quad \text{successor value}
\end{align*}
\]

Values are *normal forms*: they cannot be evaluated further.
Recap: One Step Evaluation Rel. (1)

\( t \rightarrow t' \) is an *evaluation relation* on terms. Read:

- \( t \) evaluates to \( t' \) in one step.

The evaluation relation constitute an *operational semantics* for the example language.

\[
\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2 \quad \text{(E-IFTRUE)}
\]

\[
\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3 \quad \text{(E-IFFALSE)}
\]

\[
\frac{t_1 \rightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \quad \text{(E-IF)}
\]
Recap: One Step Evaluation Rel. (2)

\[
\begin{align*}
    t_1 & \rightarrow t'_1 \\
    \text{succ } t_1 & \rightarrow \text{succ } t'_1 \\
    \text{pred } 0 & \rightarrow 0 \\
    \text{pred } (\text{succ } n\nu_1) & \rightarrow n\nu_1 \\
    t_1 & \rightarrow t'_1 \\
    \text{pred } t_1 & \rightarrow \text{pred } t'_1
\end{align*}
\]

(E-SUCC)  
(E-PREDZERO)  
(E-PREDSUCC)  
(E-PRED)
Recap: One Step Evaluation Rel. (3)

\[
\begin{align*}
\text{iszero } 0 &\rightarrow \text{true } & (\text{E-ISZEROZERO}) \\
\text{iszero } (\text{succ } n v_1) &\rightarrow \text{false } & (\text{E-ISZEROSUCC}) \\
\end{align*}
\]
Recap: One Step Evaluation Rel. (4)

Evaluation of:

\[
\text{if } (\text{iszero } (\text{pred } (\text{succ } 0))) \text{ then } (\text{pred } 0) \text{ else } (\text{succ } 0)
\]
Recap: One Step Evaluation Rel. (4)

Evaluation of:

\[
\text{if (iszero (pred (succ 0))) then (pred 0) else (succ 0)}
\]

Step 1:

\[
\begin{align*}
\text{pred (succ 0)} & \rightarrow 0 & \text{E-PREDSUCC} \\
\text{iszero (pred (succ 0))} & \rightarrow \text{iszero 0} & \text{E-ISZERO} \\
\text{if (iszero (pred (succ 0))) then (pred 0) else (succ 0)} & \rightarrow \text{if (iszero 0) then (pred 0) else (succ 0)} & \text{E-IF}
\end{align*}
\]
Step 2:

\[
\text{iszero } 0 \rightarrow \text{true} \quad \text{E-ISZERZERO}
\]

\[
\text{if (iszero } 0\text{) then (pred } 0\text{) else (succ } 0\text{)}
\]

\[
\rightarrow \text{if true then (pred } 0\text{) else (succ } 0\text{)}
\]
Recap: One Step Evaluation Rel. (5)

Step 2:

\[
\text{if (iszero 0) then (pred 0) else (succ 0)} 
\rightarrow \text{if true then (pred 0) else (succ 0)}
\]

Step 3:

\[
\text{if true then (pred 0) else (succ 0)} \rightarrow \text{pred 0}
\]
Recap: One Step Evaluation Rel. (5)

Step 2:

\[
\text{iszero } 0 \rightarrow \text{true} \quad \text{E-ISZEROZERO} \\
\text{if } (\text{iszero } 0) \text{ then } (\text{pred } 0) \text{ else } (\text{succ } 0) \rightarrow \text{if true then (pred } 0) \text{ else (succ } 0) \\
\]

Step 3:

\[
\text{if true then (pred } 0) \text{ else (succ } 0) \rightarrow \text{pred } 0 \quad \text{E-IFTRUE} \\
\]

Step 4:

\[
\text{pred } 0 \rightarrow 0 \quad \text{E-PREDZERO} \\
\]
Certain “obviously nonsensical” states are stuck: the term cannot be evaluated further, but it is not a value. For example:

```plaintext
if 0 then pred 0 else 0
```
Certain “obviously nonsensical” states are *stuck*: the term cannot be evaluated further, but it is *not a value*. For example:

\[
\text{if } 0 \text{ then } \text{pred } 0 \text{ else } 0
\]

Definition: A term is *stuck* if it is a normal form but not a value.
Stuck Terms (1)

- Certain “obviously nonsensical” states are **stuck**: the term cannot be evaluated further, but it is **not a value**. For example:
  
  ```
  if 0 then pred 0 else 0
  ```

- Definition: A term is **stuck** if it is a normal form but not a value.

- Why stuck???
Stuck Terms (1)

- Certain “obviously nonsensical” states are **stuck**: the term cannot be evaluated further, but it is **not a value**. For example:
  
  ```
  if 0 then pred 0 else 0
  ```

- Definition: A term is **stuck** if it is a normal form but not a value.

- Why stuck???
  - The program is **not well-defined** according to the dynamic semantics of the language.
Stuck Terms (1)

- Certain “obviously nonsensical” states are **stuck**: the term cannot be evaluated further, but it is **not a value**. For example:
  ```latex
  \text{if } 0 \text{ then pred 0 else 0}
  ```

- Definition: A term is **stuck** if it is a normal form but not a value.

- Why stuck???
  - The program is **not well-defined** according to the dynamic semantics of the language.
  - We are attempting to **break the abstractions** of the language.
We let the notion of getting stuck model *run-time errors.*
Recap: Type Systems

Definitions (Pierce):

• A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.
Recap: Type Systems

Definitions (Pierce):

• A **type system** is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

• A **safe language** is one that protects its abstractions.
Recap: Type Systems

Definitions (Pierce):

- A **type system** is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

- A **safe language** is one that protects its abstractions.

Our goal is thus a type system that rules out semantically ill-defined programs, i.e. that guarantees that a program never gets stuck!
Why Should We Care About Safety?
Why Should We Care About Safety?

• One reason: security.
Why Should We Care About Safety?

- One reason: security.
- C/C++ is unsafe: buffer overruns possible.
Why Should We Care About Safety?

- One reason: security.
- C/C++ is unsafe: buffer overruns possible.
- Buffer overruns allows input data to be executed as code.
Why Should We Care About Safety?

- One reason: security.
- C/C++ is unsafe: buffer overruns possible.
- Buffer overruns allows input data to be executed as code.
- One of the most common security holes: Had a safe variant of C been used, one might speculate that billions of dollars would have been saved.
Why Should We Care About Safety?

- One reason: security.
- C/C++ is unsafe: buffer overruns possible.
- Buffer overruns allows input data to be executed as code.
- One of the most common security holes: Had a safe variant of C been used, one might speculate that billions of dollars would have been saved.

Today, we’re going to see how to go about proving that the *design* of a language is safe.
At this point, there are only two *types*, booleans and the natural numbers:

\[ T \rightarrow \text{types:} \]

- **Bool** \quad \text{type of booleans}
- **Nat** \quad \text{type of natural numbers}
Typing Relation

We will define a *typing relation* between terms and types:

\[ t : T \]
Typing Relation

We will define a *typing relation* between terms and types:

\[ t : T \]

Read:

\[ t \text{ has type } T \]
Typing Relation

We will define a **typing relation** between terms and types:

\[ t : T \]

Read:

\[ t \text{ has type } T \]

A term that has a type, i.e., is related to a type by such a typing relation, is said to be **well-typed**.
Typing Relation

We will define a *typing relation* between terms and types:

\[ t : T \]

Read:

\[ t \text{ has type } T \]

A term that has a type, i.e., is related to a type by such a typing relation, is said to be *well-typed*.

The typing relation will be defined by (schematic) typing rules, in the same way we defined the evaluation relation.
Typing Rules

\texttt{true : Bool} \quad \text{(T-TRUE)}
Typing Rules

true : Bool  (T-TRUE)
false : Bool (T-FALSE)
Typing Rules

true : Bool  (T-TRUE)

false : Bool (T-FALSE)

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

true : Bool (T-TRUE)

false : Bool (T-FALSE)

\[
\frac{\text{true} : \text{Bool}}{\text{false} : \text{Bool}}
\]

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

0 : Nat (T-ZERO)
Typing Rules

true : Bool \hspace{2cm} \text{(T-TRUE)}

false : Bool \hspace{2cm} \text{(T-FALSE)}

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

0 : Nat \hspace{2cm} \text{(T-ZERO)}

\[
\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \quad \text{(T-SUCC)}
\]
Typing Rules

true : Bool  \hspace{1cm} (T-TRUE)

false : Bool  \hspace{1cm} (T-FALSE)

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

0 : Nat  \hspace{1cm} (T-ZERO)

\[ \frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \]  \hspace{1cm} (T-SUCC)

\[ \frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \]  \hspace{1cm} (T-PRED)
Typing Rules

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

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

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

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

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

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

\[
\frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}} \quad \text{(T-ISZERO)}
\]
Exercise

What (if any) is the type of the following terms?

- \( \text{if} \ (\text{iszero} \ (\text{succ} \ 0)) \ \text{then} \ (\text{succ} \ 0) \ \text{else} \ 0 \)
- \( \text{if} \ 0 \ \text{then} \ \text{pred} \ 0 \ \text{else} \ 0 \)
Safety = Progress + Preservation (1)

The most basic property of a type system: safety, or "well typed programs do not go wrong", where "wrong" means entering a "stuck state".

This breaks down into two parts:
Safety = Progress + Preservation (1)

The most basic property of a type system: safety, or “well typed programs do not go wrong”, where “wrong” means entering a “stuck state”.

This breaks down into two parts:

- **Progress**: A well-typed term is not stuck.
Safety = Progress + Preservation (1)

The most basic property of a type system: safety, or “well typed programs do not go wrong”, where “wrong” means entering a “stuck state”.

This breaks down into two parts:

- **Progress**: A well-typed term is not stuck.
- **Preservation**: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.
Safety = Progress + Preservation (1)

The most basic property of a type system: safety, or “well typed programs do not go wrong”, where “wrong” means entering a “stuck state”.

This breaks down into two parts:

- **Progress**: A well-typed term is not stuck.
- **Preservation**: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

Together, these two properties say that a well-typed term can never reach a stuck state during evaluation.
Safety = Progress + Preservation (2)

Formally:

- **THEOREM [PROGRESS]:** Suppose that $t$ is a well-typed term (i.e., $t : T$), then either $t$ is a value or else there is some $t'$ such that $t \rightarrow t'$.

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

Safety = Progress + Preservation (2)

Formally:

- **THEOREM [PROGRESS]:** Suppose that $t$ is a well-typed term (i.e., $t : T$), then either $t$ is a value or else there is some $t'$ such that $t \rightarrow t'$.

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

- **THEOREM [PRESERVATION]:**
  If $t : T$ and $t \rightarrow t'$, then $t' : T$.

  **PROOF:** By induction on a derivation of $t : T$. 
Safety = Progress + Preservation (2)

Formally:

- **THEOREM [PROGRESS]**: Suppose that $t$ is a well-typed term (i.e., $t : T$), then either $t$ is a value or else there is some $t'$ such that $t \rightarrow t'$.
  
  **PROOF**: By induction on a derivation of $t : T$.

- **THEOREM [PRESERVATION]**: If $t : T$ and $t \rightarrow t'$, then $t' : T$.
  
  **PROOF**: By induction on a derivation of $t : T$.

  (Strong form: exact type $T$ preserved.)
Progress: A Proof Fragment (1)

The relevant *typing* and *evaluation* rules for the case T-IF:

\[
\begin{align*}
\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} & \quad \text{(T-IF)} \\
\frac{\text{if } \text{true then } t_2 \text{ else } t_3 \longrightarrow t_2}{(E-IFTRUE)} \\
\frac{\text{if } \text{false then } t_2 \text{ else } t_3 \longrightarrow t_3}{(E-IFFALSE)} \\
\frac{t_1 \longrightarrow t_1'}{\frac{\text{if } t_1 \text{ then } t_2 \text{ else } t_3}{\text{if } t_1' \text{ then } t_2 \text{ else } t_3}} & \quad \text{(E-IF)}
\end{align*}
\]
A typical case when proving Progress by induction on a derivation of $t : T$.

Case T-IF: $t = \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3$

$t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$
Progress: A Proof Fragment (2)

A typical case when proving Progress by induction on a derivation of \( t : T \).

Case T-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 ind. hyp, either \( t_1 \) is a value, or else there is some \( t'_1 \) such that \( t_1 \rightarrow t'_1 \).
A typical case when proving Progress by induction on a derivation of $t : T$.

Case T-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 ind. hyp, 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 it must be either $\text{true}$ or $\text{false}$, in which case either $E\text{-IFTRUE}$ or $E\text{-IFFFALSE}$ applies to $t$. 
A typical case when proving Progress by induction on a derivation of \( t : T \).

Case T-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 ind. hyp, 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 it must be either \text{true} or \text{false}, in which case either \( \text{E-IFTRUE} \) or \( \text{E-IFFALSE} \) applies to \( t \). On the other hand, if \( t_1 \rightarrow t'_1 \), then by \( \text{E-IF} \), \( t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \).
Exceptions (1)

What about terms like

- division by zero
- head of empty list

that usually are considered well-typed?
Exceptions (1)

What about terms like

- division by zero
- head of empty list

that usually are considered well-typed?

If the type system does not rule them out, we need to differentiate those from stuck terms, or we can no longer claim that “well-typed programs do not go wrong”!
Exceptions (2)

Idea: allow *exceptions* to be raised, and make it well-defined what happens when exceptions are raised.
Exceptions (2)

Idea: allow *exceptions* to be raised, and make it well-defined what happens when exceptions are raised.

For example:

- introduce a term *error*
Exceptions (2)

Idea: allow *exceptions* to be raised, and make it well-defined what happens when exceptions are raised.

For example:

- introduce a term *error*
- introduce evaluation rules like

\[
\text{head } [] \rightarrow \text{error}
\]
Exceptions (2)

Idea: allow *exceptions* to be raised, and make it well-defined what happens when exceptions are raised.

For example:

- introduce a term `error`
- introduce evaluation rules like
  
  \[
  \text{head} \[\] \rightarrow \text{error}
  \]
- typing rule: `error : T`
Exceptions (3)

- introduce propagation rules to ensure that the entire program evaluates to error once the exception has been raised (unless there is some exception handling mechanism), e.g.:

\[ \text{pred error} \rightarrow \text{error} \]
Exceptions (3)

- introduce propagation rules to ensure that the entire program evaluates to \texttt{error} once the exception has been raised (unless there is some exception handling mechanism), e.g.:

\[
\text{pred error} \rightarrow \text{error}
\]

- change the Progress theorem slightly to allow for exceptions:

THEOREM [PROGRESS]: Suppose that \( t \) is a well-typed term (i.e., \( t : T \)), then either \( t \) is a value or \texttt{error}, or else there is some \( t' \) with \( t \rightarrow t' \).
Extension: Let-bound Variables (1)

Syntactic extension:

\[
\begin{align*}
t & \to \ldots & \text{terms:} \\
| & x & \text{variable} \\
| & \textbf{let } x = t \textbf{ in } t & \text{let-expression}
\end{align*}
\]

New evaluation rules:

\[
\begin{align*}
\textbf{let } x = v_1 \textbf{ in } t_2 & \longrightarrow [x \mapsto v_1]t_2 & \text{(E-LETV)} \\
\frac{t_1 \longrightarrow t'_1}{\textbf{let } x = t_1 \textbf{ in } t_2 \longrightarrow \textbf{let } x = t'_1 \textbf{ in } t_2} & \text{(E-LET)}
\end{align*}
\]
We now need a **typing context** or **type environment** to keep track of types of variables (an abstract version of a “symbol table”).

The typing relation thus becomes a **ternary relation**:

\[ \Gamma \vdash t : T \]

Read: term \( t \) has type \( T \) in type environment \( \Gamma \).
Environment-related notation:

- Extending an environment:
  \[ \Gamma, x : T \]
  The new declaration is understood to replace any earlier declaration for a variable with the same name.

- Stating that the type of a variable is given by an environment:
  \[ x : T \in \Gamma \quad \text{or} \quad \Gamma(x) = T \]
Extension: Let-bound Variables (4)

Updated typing rules:

\[ \Gamma \vdash \text{true} : \text{Bool} \]  
\[ \Gamma \vdash \text{false} : \text{Bool} \]  
\[ \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \]  
  
\[ (\text{T-TRUE}) \]  
\[ (\text{T-FALSE}) \]  
\[ (\text{T-IF}) \]
Extension: Let-bound Variables (5)

Updated typing rules:

\[ \Gamma \vdash 0 : \text{Nat} \quad \text{(T-ZERO)} \]

\[ \Gamma \vdash t_1 : \text{Nat} \quad \text{(T-SUCC)} \]

\[ \Gamma \vdash \text{succ } t_1 : \text{Nat} \]

\[ \Gamma \vdash t_1 : \text{Nat} \quad \text{(T-PRED)} \]

\[ \Gamma \vdash \text{pred } t_1 : \text{Nat} \]

\[ \Gamma \vdash t_1 : \text{Nat} \quad \text{(T-ISZERO)} \]

\[ \Gamma \vdash \text{iszero } t_1 : \text{Bool} \]
Extension: Let-bound Variables (6)

New typing rules:

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (T\text{-VAR})
\]
Extension: Let-bound Variables (6)

New typing rules:

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}
\]

\[
\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(T-LET)}
\]
Extension: Let-bound Variables (7)

Recursive bindings?
Recursive bindings?

Typing is straightforward if the recursively-defined entity is *explicitly* typed:

\[
\Gamma, x : T_1 \vdash t_1 : T_1 \\
\Gamma, x : T_1 \vdash t_2 : T_2 \\
\Gamma \vdash \text{let } x : T_1 = t_1 \text{ in } t_2 : T_2 
\]  

(T-LET)
Recursive bindings?

Typing is straightforward if the recursively-defined entity is *explicitly* typed:

\[
\frac{\Gamma, x : T_1 \vdash t_1 : T_1}{\Gamma \vdash \text{let } x : T_1 = t_1 \text{ in } t_2 : T_2} \quad (T\text{-LET})
\]

If not, the question is if \(T_1\) is uniquely defined (and in a type checker how to compute this type):

\[
\frac{\Gamma, x : T_1 \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad (T\text{-LET})
\]
Extension: Let-bound Variables (7)

Recursive bindings?

Typing is straightforward if the recursively-defined entity is \textit{explicitly} typed:

\[
\frac{\Gamma, x : T_1 \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x : T_1 = t_1 \text{ in } t_2 : T_2} \quad \text{(T-LET)}
\]

If not, the question is if \( T_1 \) is uniquely defined (and in a type checker how to compute this type):

\[
\frac{\Gamma, x : T_1 \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(T-LET)}
\]

\textit{(Evaluation} is more involved: we leave that for now.)
Extension: Functions (1)

Syntactic extension:

\[
\begin{align*}
t & \rightarrow \ldots & \text{terms:} \\
& | \quad \lambda x : T . t & \text{abstraction} \\
& | \quad t \ t & \text{application} \\
\end{align*}
\]

\[
\begin{align*}
v & \rightarrow \ldots & \text{values:} \\
& | \quad \lambda x : T . t & \text{abstraction value} \\
\end{align*}
\]

\[
\begin{align*}
T & \rightarrow \ldots & \text{types:} \\
& | \quad T \rightarrow T & \text{type of functions} \\
\end{align*}
\]
Extension: Functions (2)

New evaluation rules:

\[
\begin{align*}
    t_1 & \rightarrow t_1' \\
    t_1 t_2 & \rightarrow t_1' t_2 \\
    t_2 & \rightarrow t_2' \\
    v_1 t_2 & \rightarrow v_1 t_2'
\end{align*}
\]

(E-APP1)  
(E-APP2)

\[
(\lambda x : T_{11} . t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}
\]

(E-APPABS)

Note:

- left to right evaluation order: first the function (E-APP1), then the argument (E-APP2)
- **call-by-value**: the argument fully evaluated before function “invoked” (E-APPABS).
Extension: Functions (3)

New typing rules:

\[
\Gamma, x : T_1 \vdash t_2 : T_2 \\
\Gamma \vdash \lambda x : T_1 \cdot t_2 : T_1 \rightarrow T_2
\]

(T-ABS)
Extension: Functions (3)

New typing rules:

\[
\begin{align*}
\Gamma, x : T_1 & \vdash t_2 : T_2 \\
\Gamma & \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \\
\Gamma & \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma & \vdash t_2 : T_{11} \\
\Gamma & \vdash t_1 \ t_2 : T_{12}
\end{align*}
\]

(T-ABS)
(T-APP)