CS 320: Principles of Programming Languages

Katie Casamento, Portland State University

Winter 2018
Week 8a: Simply-Typed Lambda Calculus & Big Step Semantics

Types

A type categorizes expressions: it can be thought of as a **set** of expressions.

```
true                               ∈ bool
(if true then false else true)     ∈ bool

0        ∈ num
(1 + 3)  ∈ num

(λ(x: num). x)      ∈ num → num
(λ(x: num). x + 1)  ∈ num → num
```

Working with types

A *statically-typed* language has type information that can be checked before runtime.

- A *typechecking* procedure decides whether or not a typed expression has some given type
- A *type inference* procedure attempts to find a type for an untyped expression

A *dynamically-typed* language has type information that can only be checked at runtime.

- *Dynamic type checks* can be inserted by a compiler/interpreter to check the types of expressions at runtime before evaluating them

Simply-typed lambda calculus (STLC)

Simply-typed lambda calculus

- Introduced by Alonzo Church in 1940[1]
- Has interpretations in many domains
  - In PL theory as a programming language
  - In proof theory[2] as a proof system for intiuitionistic propositional logic
  - In category theory[3] as the internal language of Cartesian closed categories
- Foundation of *statically-typed* functional programming languages
- Multiple styles of presentation
  - *Intrinsic* (or *Church-style*): types are structurally part of terms
    - This is what we'll use in this class
  - *Extrinsic* (or *Curry-style*): types are a tool to analyze untyped terms

1. "A formulation of the simple theory of types"
2. "The formulae-as-types notion of construction" (William Howard, 1969)
3. "Cartesian closed categories and typed lambda-calculi" (Joachim Lambek, 1985)

Simply-typed lambda calculus

The expression syntax is just a slight modification from untyped lambda calculus:

$$
\begin{aligned}
\texttt{e} ::=\ & \texttt{x} \mid (\lambda(\texttt{x}: \texttt{t}).\ \texttt{e}) \mid (\texttt{e}_1\ \texttt{e}_2) \\
\mid\ & \texttt{n} \mid (\texttt{e}_1 + \texttt{e}_2) \mid (\texttt{e}_1 * \texttt{e}_2) \\
\mid\ & \texttt{b} \mid \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3
\end{aligned}
$$

The only difference is that lambda arguments are annotated with a *type*:

$$\texttt{t} ::= \texttt{bool} \mid \texttt{num} \mid \texttt{t}_1 \to \texttt{t}_2$$

The arrow is right-associative, so "$\texttt{t}_1 \to \texttt{t}_2 \to \texttt{t}_3$" reads as "$\texttt{t}_1 \to (\texttt{t}_2 \to \texttt{t}_3)$".

The arrow is for *function types* (like in Haskell): "$\texttt{t}_1 \to \texttt{t}_2$" is the type of functions with argument (or *domain*) type $\texttt{t}_1$ and return (or *codomain*) type $\texttt{t}_2$.

Primitive typing rules

$$
\begin{aligned}
\texttt{e} ::=\ & \texttt{x} \mid (\lambda(\texttt{x}: \texttt{t}).\ \texttt{e}) \mid (\texttt{e}_1\ \texttt{e}_2) \\
\mid\ & \texttt{n} \mid (\texttt{e}_1 + \texttt{e}_2) \mid (\texttt{e}_1 * \texttt{e}_2) \\
\mid\ & \texttt{b} \mid \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 \\
\texttt{t} ::=\ & \texttt{bool} \mid \texttt{num} \mid \texttt{t}_1 \to \texttt{t}_2
\end{aligned}
$$

To specify the types of expressions, we use *typing rules*.

$$\texttt{T-Bool}\quad \frac{}{\texttt{b} : \texttt{bool}}$$

(under any condition)

a Boolean constant `b` has type `bool`

$$\texttt{T-Num}\quad \frac{}{\texttt{n} : \texttt{num}}$$

(under any condition)

a Boolean constant `n` has type `num`

Numeric operation typing rules

$$
\begin{aligned}
\texttt{e} ::=\ & \texttt{x} \mid (\lambda(\texttt{x}: \texttt{t}).\ \texttt{e}) \mid (\texttt{e}_1\ \texttt{e}_2) \\
\mid\ & \texttt{n} \mid (\texttt{e}_1 + \texttt{e}_2) \mid (\texttt{e}_1 * \texttt{e}_2) \\
\mid\ & \texttt{b} \mid \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 \\
\texttt{t} ::=\ & \texttt{bool} \mid \texttt{num} \mid \texttt{t}_1 \to \texttt{t}_2
\end{aligned}
$$

$$\texttt{T-Plus}\quad \frac{\texttt{e}_1 : \texttt{num} \qquad \texttt{e}_2 : \texttt{num}}{(\texttt{e}_1 + \texttt{e}_2) : \texttt{num}}$$

if $\texttt{e}_1$ has type `num` and $\texttt{e}_2$ has type `num`,

then $(\texttt{e}_1 + \texttt{e}_2)$ has type `num`

$$\texttt{T-Times}\quad \frac{\texttt{e}_1 : \texttt{num} \qquad \texttt{e}_2 : \texttt{num}}{(\texttt{e}_1 * \texttt{e}_2) : \texttt{num}}$$

if $\texttt{e}_1$ has type `num` and $\texttt{e}_2$ has type `num`,

then $(\texttt{e}_1 * \texttt{e}_2)$ has type `num`

Type derivations

$$\text{T-Plus}\ \frac{e_1 : \text{num} \quad e_2 : \text{num}}{(e_1 + e_2) : \text{num}} \qquad \text{T-Times}\ \frac{e_1 : \text{num} \quad e_2 : \text{num}}{(e_1 * e_2) : \text{num}} \qquad \text{T-Num}\ \frac{}{n : \text{num}}$$

In order to prove that a term has some type, we construct a *type derivation* by combining type rules in a tree structure.

A derivation is *valid* if every horizontal line corresponds to a valid typing rule, and *complete* if the top of every branch of the tree is a true logical statement (or empty).

A term that can be given a complete and valid type derivation is *well-typed*.

$$\text{T-Plus}\ \frac{\text{T-Num}\ \dfrac{}{1 : \text{num}} \qquad \text{T-Times}\ \dfrac{\text{T-Num}\ \dfrac{}{2 : \text{num}} \quad \text{T-Num}\ \dfrac{}{3 : \text{num}}}{(2 * 3) : \text{num}}}{(1 + (2 * 3)) : \text{num}}$$

If/then/else typing rule

```
e ::= x | (λ(x: t). e) | (e₁ e₂)
    | n | (e₁ + e₂) | (e₁ * e₂)
    | b | if e₁ then e₂ else e₃
t ::= bool | num | t₁ → t₂
```

Both branches of an if/then/else expression must have the same type.

$$\text{T-If}\ \frac{e_1 : \text{bool} \quad e_2 : t \quad e_3 : t}{(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : t}$$

if e_1 has type `bool` and e_2/e_3 have the same type t, then $(\text{if } e_1 \text{ then } e_2 \text{ else } e_3)$ has type t

Otherwise, we wouldn't know which type the whole if/then/else should have.

```
1 : num
true : bool
if true then 1 else true : ???
```

Application typing rule

```
e ::= x | (λ(x: t). e) | (e₁ e₂)
    | n | (e₁ + e₂) | (e₁ * e₂)
    | b | if e₁ then e₂ else e₃
t ::= bool | num | t₁ → t₂
```

The application rule requires that the function's input type is the same as the type of the argument being passed to it.

$$\text{T-App}\ \frac{e_1 : t_1 \to t_2 \quad e_2 : t_1}{(e_1\ e_2) : t_2}$$

if e_1 has type $t_1 \to t_2$ and e_2 has type t_1, then $(e_1\ e_2)$ has type t_2

Variable typing rules

```
e ::= x | (λ(x: t). e) | (e₁ e₂)
    | n | (e₁ + e₂) | (e₁ * e₂)
    | b | if e₁ then e₂ else e₃
t ::= bool | num | t₁ → t₂
```

What type should a **variable** have?

$$\text{T-Var}\ \frac{}{x : ???}$$

$(\lambda(x : \text{num}).\ x)$ — $x : \text{num}$

$(\lambda(x : \text{bool}).\ x)$ — $x : \text{bool}$

It depends on the *context* of the variable!

Variable typing rules

$$\begin{array}{rl} e ::= & x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ & \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ & \mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ t ::= & \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{array}$$

What type should a **variable** have?

It depends on the *context* of the variable!

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

if x is mapped to type t in context Γ,

then x has type t under Γ
(often pronounced "gamma entails x is of type t")

Contexts

$$\begin{array}{rl} e ::= & x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ & \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ & \mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ t ::= & \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{array}$$

A *type environment* (or *context*) is an **ordered list of mappings** from variables to types.

For example:

```
a: num, b: bool
a: num, f: bool → num, a: bool, b: num
```

The syntax "$\Gamma(a)\ =\ t$" means "the leftmost occurrence of a in Γ is mapped to t".

```
(a: num, b: bool)(a) = num
(a: num, b: bool)(b) = bool
(a: num, f: bool → num, a: bool, b: num)(a) = num
```

The symbol $\varnothing$ is sometimes used to represent the empty context.

A type derivation with variables

$$\begin{array}{rl} e ::= & x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ & \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ & \mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ t ::= & \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{array}$$

A derivation is *valid* if every horizontal line corresponds to a valid typing rule, and *complete* if the top of every branch of the tree is a true statement (or empty).

$$\text{T-Var}\quad \frac{\Gamma(x)\ =\ t}{\Gamma \vdash x : t} \qquad\qquad \text{T-Num}\quad \frac{}{\Gamma \vdash n : \texttt{num}}$$

$$\text{T-If}\quad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) : t}$$

$$\text{T-If}\quad \frac{\text{T-Var}\ \dfrac{(x{:}\ \texttt{bool})(x)\ =\ \texttt{bool}}{x{:}\ \texttt{bool} \vdash x : \texttt{bool}} \quad \text{T-Num}\ \dfrac{}{x{:}\ \texttt{bool} \vdash 1 : \texttt{num}} \quad \text{T-Num}\ \dfrac{}{x{:}\ \texttt{bool} \vdash 0 : \texttt{num}}}{x{:}\ \texttt{bool} \vdash (\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ 0) : \texttt{num}}$$

Lambda typing rule

$$\begin{array}{rl} e ::= & x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ & \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ & \mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ t ::= & \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{array}$$

What type should a **lambda** have?

$$\text{T-Lam}\quad \frac{???}{\Gamma \vdash (\lambda(x{:}\ t_1).\ e) : t_1 \to ???}$$

We know the argument type from the syntax, but the return type is determined by the **body** of the lambda.

Lambda typing rule

$$e ::= x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$
$$t ::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2$$

What type should a **lambda** have?

$$\text{T-Lam}\ \frac{x{:}\ t_1,\ \Gamma \vdash e : t_2}{\Gamma \vdash (\lambda(x{:}\ t_1).\ e) : t_1 \to t_2}$$

if e has type t_2 under Γ extended with $x{:}\ t_1$, then $(\lambda(x{:}\ t_1).\ e)$ has type $t_1 \to t_2$

The body is typechecked under a context **extended** with a mapping for the argument.

A type derivation with lambdas

$$e ::= x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$
$$t ::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2$$

$$\text{T-Var}\ \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \text{T-Plus}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash (e_1 + e_2) : \texttt{num}}$$

$$\text{T-Lam}\ \frac{x{:}\ t_1,\ \Gamma \vdash e : t_2}{\Gamma \vdash (\lambda(x{:}\ t_1).\ e) : t_1 \to t_2}$$

$$\text{T-Lam}\ \cfrac{\text{T-Lam}\ \cfrac{\text{T-Plus}\ \cfrac{\text{T-Var}\ \cfrac{(x{:}\ \texttt{num},\ y{:}\ \texttt{num})(x) = \texttt{num}}{x{:}\ \texttt{num},\ y{:}\ \texttt{num} \vdash x : \texttt{num}} \qquad \cfrac{(x{:}\ \texttt{num},\ y{:}\ \texttt{num})(y) = \texttt{num}}{x{:}\ \texttt{num},\ y{:}\ \texttt{num} \vdash y : \texttt{num}}\ \text{T-Var}}{x{:}\ \texttt{num},\ y{:}\ \texttt{num} \vdash x + y : \texttt{num}}}{x{:}\ \texttt{num} \vdash (\lambda(y{:}\ \texttt{num}).\ x + y) : \texttt{num} \to \texttt{num}}}{\varnothing \vdash (\lambda(x{:}\ \texttt{num})\ (y{:}\ \texttt{num}).\ x + y) : \texttt{num} \to (\texttt{num} \to \texttt{num})}$$

All typing rules (so far)

$$e ::= x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$
$$t ::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2$$

$$\text{T-Bool}\ \frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \text{T-Num}\ \frac{}{\Gamma \vdash n : \texttt{num}} \qquad \text{T-Var}\ \frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

$$\text{T-Plus}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash (e_1 + e_2) : \texttt{num}} \qquad \text{T-Times}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash (e_1 * e_2) : \texttt{num}}$$

$$\text{T-If}\ \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) : t}$$

$$\text{T-Lam}\ \frac{x{:}\ t_1,\ \Gamma \vdash e : t_2}{\Gamma \vdash (\lambda(x{:}\ t_1).\ e) : t_1 \to t_2} \qquad \text{T-App}\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}$$

Static type safety (small-step)

That's a lot to keep track of. What do we get out of it?

"Well-typed terms do not **go wrong**."[1]

The property of *static type safety* involves two guarantees about evaluation:

- A *progress theorem* proves that a well-typed term either is a value or can take another step of evaluation (i.e. that it **doesn't get stuck**).
- A *preservation theorem* proves that if $e : t$ and $e \Rightarrow e'$, then $e' : t$ (i.e. taking a step of evaluation **doesn't change the type** of an expression).

These two theorems combine into a very useful property: **a well-typed term always normalizes to a value of the same type** (or diverges).

1. "Types and Programming Languages", Benjamin Pierce

Static types vs. dynamic types

Benefits of static typing

That's a lot to keep track of. What do we get out of it?

- Often easier to **prove** desirable properties of well-typed programs
- Programmers can use types as **contracts** enforced by the typechecker
- Type information can aid in **comprehension** when reading programs
- Type information gives **static analysis** tools more information to work with
- Typechecking can aid in **refactoring** by highlighting code that needs updates

Untypable terms

What do we give up in exchange for type safety?

Are there any expressions that don't get stuck during normalization, but can't be given a type?

Untypable terms (ambiguous terms)

What do we give up in exchange for type safety?

Are there any expressions that don't get stuck during normalization, but can't be given a type?

```
  (if true then 1 else false) + 1
⇒ 1 + 1
⇒ 2
```

$$\text{T-If}\ \frac{e_1 : \texttt{bool} \quad e_2 : t \quad e_3 : t}{(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) : t}$$

$$\text{T-If}\ \frac{\varnothing \vdash \texttt{true} : \texttt{bool} \qquad \varnothing \vdash 1 : \texttt{num} \qquad \varnothing \vdash \texttt{false} : \texttt{bool}}{\varnothing \vdash (\texttt{if x then 1 else false}) + 1 : \texttt{num}}$$

Untypable terms (divergent terms)

What do we give up in exchange for type safety?

Are there any expressions that don't get stuck during normalization, but can't be given a type?

$$\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$
$$\Omega \Rightarrow \Omega \Rightarrow \Omega \Rightarrow \ldots$$

$$\text{T-App}\ \frac{e_1 : t_1 \to t_2 \qquad e_2 : t_1}{(e_1\ e_2) : t_2}$$

$$\text{T-App}\ \frac{\varnothing \vdash (\lambda x.\ x\ x) : {\color{red} t_1 \to t_2} \qquad \varnothing \vdash (\lambda x.\ x\ x) : {\color{red} t_1}}{\varnothing \vdash (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) : t_2}$$

Untypable terms

What do we give up in exchange for type safety?

We rule out some potentially useful terms!

Simply-typed lambda calculus is **strongly normalizing** in this form: every expression normalizes in a finite number of steps. (So a well-typed term never diverges!)

We can recover nontermination by adding a primitive `fix : (t → t) → t` such that `Ω = fix (λx. x)`,

but in general there will almost always be some computationally valid terms that can't be given a type.

Drawbacks of static typing

Why not always use static types?

- Type systems almost never have full **coverage**
  - There will be some computationally valid terms that can't be given types
- Type systems often require some amount of **syntactic overhead**
  - The programmer must write more code in order to achieve the same result
  - Type inference can help
- Statically-typed programs are sometimes harder to **extend** with unforeseen functionality
  - Types enforce a contract, but the contract might need to change over time
  - More expressive type systems can help

Dynamic types

A *dynamically-typed* language has a notion of types at **runtime**, but not necessarily at compile time.

- *Dynamic type errors* are thrown when a program tries to evaluate an expression with invalid types at runtime
  - "`(if true then 1 else false) + 1`" evaluates without error
  - "`(if false then 1 else false) + 1`" throws a dynamic type error
- Common in languages designed for scripting and rapid prototyping
  - Python, Ruby, Javascript, Lua, Scheme, …
- *Gradual typing* is a paradigm that lets programmers prototype with dynamically-typed code and add static types incrementally
  - Typescript, Typed Racket, mypy (Python), …

Big-step semantics

$$\begin{aligned} e &::= x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ &\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ &\mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{aligned}$$

Big-step semantics

Like small-step semantics, big-step semantics specify the behavior of a program in terms of an abstract machine that operates on ASTs.

Instead of specifying each step of the machine, we define rules that specify what the **final result** of evaluating a term is.

There are a couple benefits over small-step semantics:

- Less rules to deal with
- Straightforward implementation in code as a recursive evaluation function
- Evaluation order is left implicit
- Easier to prove some things about
  - Like **type safety**!

$$\begin{aligned} e &::= x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ &\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ &\mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{aligned}$$

Big-step semantics

The typing rules we've covered hint at a set of evaluation rules:

$$\text{T-Num}\ \frac{}{\Gamma \vdash n : \texttt{num}} \qquad \text{T-Plus}\ \frac{\Gamma \vdash e_1 : \texttt{num} \qquad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash (e_1 + e_2) : \texttt{num}}$$

$$\text{E-Num}\ \frac{}{\langle\rho,\ n\rangle \Downarrow n} \qquad \text{E-Plus}\ \frac{\langle\rho,\ e_1\rangle \Downarrow n_1 \qquad \langle\rho,\ e_2\rangle \Downarrow n_2 \qquad n_1 + n_2 = n}{\langle\rho,\ (e_1 + e_2)\rangle \Downarrow n}$$

$e \Downarrow v$ means "expression e evaluates to value v".

$$\begin{aligned} e &::= x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ &\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ &\mid b \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{aligned}$$

Big-step arithmetic example

$$\text{E-Num}\ \frac{}{\langle\rho,\ n\rangle \Downarrow n} \qquad \text{E-Plus}\ \frac{\langle\rho,\ e_1\rangle \Downarrow n_1 \qquad \langle\rho,\ e_2\rangle \Downarrow n_2 \qquad n_1 + n_2 = n}{\langle\rho,\ (e_1 + e_2)\rangle \Downarrow n}$$

$$\text{E-Plus}\ \frac{\text{E-Num}\ \dfrac{}{\langle\rho,\ 1\rangle \Downarrow 1} \qquad \text{E-Plus}\ \dfrac{\text{E-Num}\ \dfrac{}{\langle\rho,\ 1\rangle \Downarrow 1} \qquad \dfrac{}{\langle\rho,\ 1\rangle \Downarrow 1}\ \text{E-Num} \qquad 1 + 1 = 2}{\langle\rho,\ (1 + 1)\rangle \Downarrow 2} \qquad 1 + 2 = 3}{\langle\rho,\ (1 + (1 + 1))\rangle \Downarrow 3}$$

Big-step variable semantics

$e ::= x \mid (\lambda(x\colon t).\ e) \mid (e_1\ e_2)$
$\quad \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$
$\quad \mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$
$t ::= \texttt{bool} \mid \texttt{num} \mid t_1 \rightarrow t_2$

How do we evaluate a variable?

We need an *evaluation environment* mapping variables to values.

$$\text{T-Var}\ \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \text{E-Var}\ \frac{\rho(x) = v}{\langle\rho,\ x\rangle \Downarrow v}$$

$\langle\rho,\ x\rangle \Downarrow v$ means "variable x evaluates to value v under the context ρ".

Like type environments, an evaluation environment is an ordered list of mappings.

$x \mapsto 1,\ y \mapsto \texttt{true}$

The $\mapsto$ symbol is usually pronounced "maps to".

Big-step variable example

$e ::= x \mid (\lambda(x\colon t).\ e) \mid (e_1\ e_2)$
$\quad \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$
$\quad \mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$
$t ::= \texttt{bool} \mid \texttt{num} \mid t_1 \rightarrow t_2$

$$\text{E-Num}\ \frac{}{\langle\rho,\ n\rangle \Downarrow n} \qquad \text{E-Plus}\ \frac{\langle\rho,\ e_1\rangle \Downarrow n_1 \quad \langle\rho,\ e_2\rangle \Downarrow n_2 \quad n_1 + n_2 = n}{\langle\rho,\ (e_1 + e_2)\rangle \Downarrow n}$$

$$\text{E-Var}\ \frac{\rho(x) = v}{\langle\rho,\ x\rangle \Downarrow v}$$

$$\text{E-Plus}\ \frac{\text{E-Var}\ \dfrac{(x \mapsto 1,\ y \mapsto 2)(x) = 1}{\langle(x \mapsto 1,\ y \mapsto 2),\ x\rangle \Downarrow 1} \quad \dfrac{(x \mapsto 1,\ y \mapsto 2)(y) = 2}{\langle(x \mapsto 1,\ y \mapsto 2),\ y\rangle \Downarrow 2}\ \text{E-Var} \quad 1 + 2 = 3}{\langle(x \mapsto 1,\ y \mapsto 2),\ (x + y)\rangle \Downarrow 3}$$

Big-step application semantics

$e ::= x \mid (\lambda(x\colon t).\ e) \mid (e_1\ e_2)$
$\quad \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$
$\quad \mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$
$t ::= \texttt{bool} \mid \texttt{num} \mid t_1 \rightarrow t_2$

How do we evaluate an application?

A first attempt (not quite right):

$$\text{T-App}\ \frac{e_1 : t_1 \rightarrow t_2 \qquad e_2 : t_1}{(e_1\ e_2) : t_2}$$

if the function e_1 evaluates, in environment ρ, to a lambda with argument x and body e'_1

and the argument e_2 evaluates, in environment ρ, to value v_2

and the body of the function lambda e'_1 evaluates, in environment ρ extended with a mapping for argument value v_2, to value v

$$\text{E-App}\ \frac{\langle\rho,\ e_1\rangle \Downarrow (\lambda x.\ e'_1) \qquad \langle\rho,\ e_2\rangle \Downarrow v_2 \qquad \langle(x \mapsto v_2,\ \rho),\ e'_1\rangle \Downarrow v}{\langle\rho,\ (e_1\ e_2)\rangle \Downarrow v}$$

then function e_1 applied to argument e_2 evaluates to value v

Big-step application semantics

$e ::= x \mid (\lambda(x\colon t).\ e) \mid (e_1\ e_2)$
$\quad \mid n \mid (e_1 + e_2) \mid (e_1 * e_2)$
$\quad \mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$
$t ::= \texttt{bool} \mid \texttt{num} \mid t_1 \rightarrow t_2$

The body of a function should be evaluated in the environment it **originated** in, not the environment it's **being used** in.

This should evaluate to 2:

$(\lambda x.\ (\lambda x.\ x + 1)\ 1)\ 3$

But with this incorrect application rule, the body x + 1 is evaluated in the outer environment containing $x \mapsto 3$.

$$\text{E-App}\ \frac{\langle\rho,\ e_1\rangle \Downarrow (\lambda x.\ e'_1) \qquad \langle\rho,\ e_2\rangle \Downarrow v_2 \qquad \langle(x \mapsto v_2,\ \rho),\ e'_1\rangle \Downarrow v}{\langle\rho,\ (e_1\ e_2)\rangle \Downarrow v}$$

Big-step lambda semantics

$$
\begin{aligned}
e &::= x \mid (\lambda(x\!:t).\ e) \mid (e_1\ e_2) \\
&\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\
&\mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2
\end{aligned}
$$

The body of a function should be evaluated in the environment it **originated** in, not the environment it's **being used** in.

A *closure* is a function paired with an environment; we say the closure *closes over* the environment that it contains.

$$\text{E-Lam}\ \frac{}{\langle \rho,\ (\lambda x.\ e)\rangle \Downarrow \langle \rho,\ (\lambda x.\ e)\rangle}$$

This is what enables *partial application*, where we can apply a single argument to a multi-argument function and get back a function of one less argument.

$$
\begin{aligned}
&\langle \varnothing,\ (\lambda x\ y.\ x + y)\rangle \Downarrow \langle \varnothing,\ (\lambda\ x\ y.\ x + y)\rangle \\
&\langle \varnothing,\ ((\lambda x\ y.\ x + y)\ 1)\rangle \Downarrow \langle (x \mapsto 1),\ (\lambda y.\ x + y)\rangle
\end{aligned}
$$

Values

$$
\begin{aligned}
e &::= x \mid (\lambda(x\!:t).\ e) \mid (e_1\ e_2) \\
&\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\
&\mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2
\end{aligned}
$$

The body of a function should be evaluated in the environment it **originated** in, not the environment it's **being used** in.

A *closure* is a function paired with an environment; we say the closure *closes over* the environment that it contains.

The set of **values** in a big-step semantics should include closures instead of lambdas.

$$v ::= x \mid x\ v_1 \ldots v_2 \mid n \mid b \mid \langle \rho,\ (\lambda x.\ e)\rangle$$

Big-step application semantics

$$
\begin{aligned}
e &::= x \mid (\lambda(x\!:t).\ e) \mid (e_1\ e_2) \\
&\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\
&\mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2
\end{aligned}
$$

The body of a function should be evaluated in the environment it **originated** in, not the environment it's **being used** in.

Now we can define the correct application rule.

if the function e_1, in environment ρ, evaluates to a closure with argument x and body e'_1 in environment ρ'

and the argument e_2, in environment ρ, evaluates to value v_2

and the body of the function lambda e'_1, in environment ρ' extended with a mapping for argument value v_2, evaluates to value v

$$\text{E-App}\ \frac{\langle \rho,\ e_1\rangle \Downarrow \langle \rho',\ (\lambda x.\ e'_1)\rangle \qquad \langle \rho,\ e_2\rangle \Downarrow v_2 \qquad \langle (x \mapsto v_2,\ \rho'),\ e'_1\rangle \Downarrow v}{\langle \rho,\ (e_1\ e_2)\rangle \Downarrow v}$$

then function e_1 applied to argument e_2 evaluates to value v

Big-step application example

$$
\begin{aligned}
e &::= x \mid (\lambda(x\!:t).\ e) \mid (e_1\ e_2) \\
&\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\
&\mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2
\end{aligned}
$$

$$\text{E-Var}\ \frac{\rho(x) = v}{\langle \rho,\ x\rangle \Downarrow v} \qquad \text{E-Num}\ \frac{}{\langle \rho,\ n\rangle \Downarrow n} \qquad \text{E-Lam}\ \frac{}{\langle \rho,\ (\lambda x.\ e)\rangle \Downarrow \langle \rho,\ (\lambda x.\ e)\rangle}$$

$$\text{E-App}\ \frac{\langle \rho,\ e_1\rangle \Downarrow \langle \rho',\ (\lambda x.\ e'_1)\rangle \qquad \langle \rho,\ e_2\rangle \Downarrow v_2 \qquad \langle (x \mapsto v_2,\ \rho'),\ e'_1\rangle \Downarrow v}{\langle \rho,\ (e_1\ e_2)\rangle \Downarrow v}$$

$$\text{E-App}\ \frac{\text{E-Lam}\ \dfrac{}{\langle \varnothing,\ (\lambda x.\ x)\rangle \Downarrow \langle \varnothing,\ (\lambda x.\ x)\rangle} \qquad \text{E-Num}\ \dfrac{}{\langle \varnothing,\ 1\rangle \Downarrow 1} \qquad \text{E-Var}\ \dfrac{(x \mapsto 1)(x) = 1}{\langle (x \mapsto 1),\ x\rangle \Downarrow 1}}{\langle \varnothing,\ ((\lambda x.\ x)\ 1)\rangle \Downarrow 1}$$

Nontermination

What does a big-step derivation for a nonterminating term look like?

$\langle \varnothing,\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))\rangle \Downarrow\ ???$

There are three assumptions to prove:

1. $\langle \rho,\ e_1\rangle \Downarrow \langle \rho',\ (\lambda x.\ e'_1)\rangle \;=\; \langle \varnothing,\ (\lambda x.\ x\ x)\rangle \Downarrow \langle \varnothing,\ (\lambda x.\ x\ x)\rangle$
2. $\langle \rho,\ e_2\rangle \Downarrow v_2 \;=\; \langle \varnothing,\ (\lambda x.\ x\ x)\rangle \Downarrow \langle \varnothing,\ (\lambda x.\ x\ x)\rangle$
3. $\langle (x \mapsto v_2,\ \rho'),\ e'_1\rangle \Downarrow v \;=\; \langle (x \mapsto \langle \varnothing,\ (\lambda x.\ x\ x)\rangle),\ (x\ x)\rangle \Downarrow\ ???$

$$\text{E-App}\ \dfrac{\overset{1.}{\langle \rho,\ e_1\rangle \Downarrow \langle \rho',\ (\lambda x.\ e'_1)\rangle} \qquad \overset{2.}{\langle \rho,\ e_2\rangle \Downarrow v_2} \qquad \overset{3.}{\langle (x \mapsto v_2,\ \rho'),\ e'_1\rangle \Downarrow v}}{\langle \rho,\ (e_1\ e_2)\rangle \Downarrow v}$$

Nontermination

What does a big-step derivation for a nonterminating term look like?

$\langle \varnothing,\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))\rangle \Downarrow\ ???$

1. $\langle \rho,\ e_1\rangle \Downarrow \langle \rho',\ (\lambda x.\ e'_1)\rangle \;=\; \langle \varnothing,\ (\lambda x.\ x\ x)\rangle \Downarrow \langle \varnothing,\ (\lambda x.\ x\ x)\rangle$
2. $\langle \rho,\ e_2\rangle \Downarrow v_2 \;=\; \langle \varnothing,\ (\lambda x.\ x\ x)\rangle \Downarrow \langle \varnothing,\ (\lambda x.\ x\ x)\rangle$
3. $\langle (x \mapsto v_2,\ \rho'),\ e'_1\rangle \Downarrow v \;=\; \langle (x \mapsto \langle \varnothing,\ (\lambda x.\ x\ x)\rangle),\ (x\ x)\rangle \Downarrow\ ???$

$$\text{E-Lam}\ \dfrac{}{\langle \rho,\ (\lambda x.\ e)\rangle \Downarrow \langle \rho,\ (\lambda x.\ e)\rangle} \qquad \text{E-Lam}\ \dfrac{}{\langle \varnothing,\ (\lambda x.\ x\ x)\rangle \Downarrow \langle \varnothing,\ (\lambda x.\ x\ x)\rangle}$$

Nontermination

What does a big-step derivation for a nonterminating term look like?

$\langle \varnothing,\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))\rangle \Downarrow\ ???$

1. $\langle \rho,\ e_1\rangle \Downarrow \langle \rho',\ (\lambda x.\ e'_1)\rangle \;=\; \langle \varnothing,\ (\lambda x.\ x\ x)\rangle \Downarrow \langle \varnothing,\ (\lambda x.\ x\ x)\rangle$
2. $\langle \rho,\ e_2\rangle \Downarrow v_2 \;=\; \langle \varnothing,\ (\lambda x.\ x\ x)\rangle \Downarrow \langle \varnothing,\ (\lambda x.\ x\ x)\rangle$
3. $\langle (x \mapsto v_2,\ \rho'),\ e'_1\rangle \Downarrow v \;=\; \langle (x \mapsto \langle \varnothing,\ (\lambda x.\ x\ x)\rangle),\ (x\ x)\rangle \Downarrow\ ???$

$$\text{E-App}\ \dfrac{\langle \rho,\ e_1\rangle \Downarrow \langle \rho',\ (\lambda x.\ e'_1)\rangle \qquad \langle \rho,\ e_2\rangle \Downarrow v_2 \qquad \langle (x \mapsto v_2,\ \rho'),\ e'_1\rangle \Downarrow v}{\langle \rho,\ (e_1\ e_2)\rangle \Downarrow v}$$

$\omega = \lambda x.\ x\ x$

$\rho = (x \mapsto \langle \varnothing,\ \omega\rangle)$

$$\text{E-App}\ \dfrac{\langle \rho,\ x\rangle \Downarrow \langle \varnothing,\ \omega\rangle \qquad \langle \rho,\ x\rangle \Downarrow \langle \varnothing,\ \omega\rangle \qquad \langle \rho,\ (x\ x)\rangle \Downarrow\ ???}{\langle \rho,\ (x\ x)\rangle \Downarrow\ ???}$$

Nontermination

What does a big-step derivation for a nonterminating term look like?

$\langle \varnothing,\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))\rangle \Downarrow\ ???$

The derivation depends on itself - it would have to be infinitely large!

Since this is illegal (the proof system is a formal language), there is **no big-step derivation** for a nonterminating term.

This is sometimes written $\langle \rho,\ e\rangle \Downarrow \bot$, where the $\bot$ symbol is pronounced "bottom".

$\omega = \lambda x.\ x\ x$

$\rho = (x \mapsto \langle \varnothing,\ \omega\rangle)$

$$\text{E-App}\ \dfrac{\langle \rho,\ x\rangle \Downarrow \langle \varnothing,\ \omega\rangle \qquad \langle \rho,\ x\rangle \Downarrow \langle \varnothing,\ \omega\rangle \qquad \langle \rho,\ (x\ x)\rangle \Downarrow\ ???}{\langle \rho,\ (x\ x)\rangle \Downarrow\ ???}$$

Big-step implementation

Big-step implementation

$$\begin{aligned} e &::= x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ &\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ &\mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\ t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{aligned}$$

The big-step rules suggest cases for a functional *evaluation function*.

$$\text{E-Num}\ \frac{}{\langle\rho,\ n\rangle \Downarrow n} \qquad \text{E-Plus}\ \frac{\langle\rho,\ e_1\rangle \Downarrow n_1 \quad \langle\rho,\ e_2\rangle \Downarrow n_2 \quad n_1 + n_2 = n}{\langle\rho,\ (e_1 + e_2)\rangle \Downarrow n}$$

In pseudo-Haskell:

```
eval :: Exp → Env → Val
…
eval "n"       ρ = n
eval "e₁ + e₂" ρ = eval e₁ ρ + eval e₂ ρ
…
```

Big-step errors

$$\begin{aligned} e &::= x \mid (\lambda(x{:}\ t).\ e) \mid (e_1\ e_2) \\ &\mid n \mid (e_1 + e_2) \mid (e_1 * e_2) \\ &\mid b \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\ t &::= \texttt{bool} \mid \texttt{num} \mid t_1 \to t_2 \end{aligned}$$

The big-step rules suggest cases for a functional *evaluation function*.

One potential problem is that this function is *partial*: it doesn't define an output for every possible input.

```
eval "true + 1" ∅ = ???
```

Like in small-step semantics, we can remedy this by adding an **error** value along with rules to reduce every invalid term to an error.

$$\text{E-PlusErrL}\ \frac{\langle\rho,\ e_1\rangle \not\Downarrow n}{\langle\rho,\ (e_1 + e_2)\rangle \Downarrow \texttt{err}} \qquad \text{E-PlusErrR}\ \frac{\langle\rho,\ e_2\rangle \not\Downarrow n}{\langle\rho,\ (e_1 + e_2)\rangle \Downarrow \texttt{err}} \qquad \text{etc.}$$

This is effectively a kind of *dynamic typechecking*.

Big-step type safety

Type safety proof sketch

To prove properties of programs, we often use *structural induction*:

- If we can prove the property for every **primitive** term,
- and we can prove the property for every **compound** term when assuming the property for all of its subterms,
- then we've proved the property for **all** terms.

In this case, the property we want to prove is:

Every well-typed expression can be given a valid and complete big-step derivation that reduces it to a value of the same type.

(Remember that well-typed expressions never diverge!)

Type safety proof sketch: primitives

Every well-typed expression can be given a valid and complete big-step derivation that results in a value of the same type as the expression.

The **primitive** cases are trivial: every primitive immediately evaluates to a value of the same type (the primitive itself).

$$\text{T-Num}\ \frac{}{\Gamma \vdash \texttt{n : num}} \qquad \text{E-Num}\ \frac{}{\langle \rho,\ \texttt{n}\rangle \Downarrow \texttt{n}}$$

$$\text{T-Bool}\ \frac{}{\Gamma \vdash \texttt{b : bool}} \qquad \text{E-Bool}\ \frac{}{\langle \rho,\ \texttt{b}\rangle \Downarrow \texttt{b}}$$

Type safety proof sketch: addition

Addition is a **compound** term, so we get to assume that type safety holds for both arguments (thanks to structural induction).

$$\text{T-Plus}\ \frac{\Gamma \vdash e_1 : \texttt{num} \quad \Gamma \vdash e_2 : \texttt{num}}{\Gamma \vdash (e_1 + e_2) : \texttt{num}}$$

$$\text{E-Plus}\ \frac{\langle \rho,\ e_1\rangle \Downarrow n_1 \quad \langle \rho,\ e_2\rangle \Downarrow n_2 \quad n_1 + n_2 = n}{\langle \rho,\ (e_1 + e_2)\rangle \Downarrow n}$$

1. e_1 reduces to some number n_1 (induction hypothesis)
2. e_2 reduces to some number n_2 (induction hypothesis)
3. There exists some number n such that $n_1 + n_2 = n$ (property of arithmetic)
4. So $(e_1 + e_2)$ reduces to some value (n) of type `num`

Type safety proof sketch: lambdas/closures

Every well-typed expression can be given a valid and complete big-step derivation that results in a value of the same type as the expression.

What is the type of a closure?

$$\text{T-Lam}\ \frac{x: t_1,\ \Gamma \vdash e : t_2}{\Gamma \vdash (\lambda(x: t_1).\ e) : t_1 \to t_2}$$

$$\text{T-Cls}\ \frac{???}{\Gamma \vdash \langle \rho,\ \lambda(x: t_1).\ e\rangle : ???}$$

The **body** of the closure should be typed with respect to the **environment** of the closure, not the context the closure appears in!

Type safety proof sketch: environments

We need a notion of a **well-typed environment**:

$$\text{T-Env}\ \frac{\forall x.\ \Gamma(x) = t \text{ implies } \rho(x) : t}{\Gamma \vdash \rho}$$

if every value in ρ is well-typed under Γ,
then ρ is well-typed under Γ

A closure's type depends only on the **type context of its environment**.

$$\text{T-Cls}\ \frac{\Gamma' \vdash \rho \qquad x{:}\ t_1, \Gamma' \vdash e : t_2}{\Gamma \vdash \langle\rho,\ (\lambda(x{:}\ t_1).\ e)\rangle : t_1 \rightarrow t_2}$$

if ρ is well-typed under some context Γ'
and e has type t_2 under Γ' extended with $x{:}\ t_1$

then $\langle\rho,\ (\lambda(x{:}\ t_1).\ e)\rangle$ has type $t_1 \rightarrow t_2$
under **any arbitrary type context** Γ

Type safety proof sketch: lambdas/closures

Now we can show type safety for lambda terms:

$$\text{T-Env}\ \frac{\forall x.\ \Gamma(x) = t \text{ implies } \rho(x) : t}{\Gamma \vdash \rho} \qquad \text{T-Cls}\ \frac{\Gamma' \vdash \rho \qquad x{:}\ t_1, \Gamma' \vdash e : t_2}{\Gamma \vdash \langle\rho,\ (\lambda(x{:}\ t_1).\ e)\rangle : t_1 \rightarrow t_2}$$

$$\text{T-Lam}\ \frac{x{:}\ t_1, \Gamma \vdash e : t_2}{\Gamma \vdash (\lambda(x{:}\ t_1).\ e) : t_1 \rightarrow t_2} \qquad \text{E-Lam}\ \frac{}{\langle\rho,\ (\lambda(x{:}\ t_1).\ e)\rangle \Downarrow \langle\rho,\ (\lambda(x{:}\ t_1).\ e)\rangle}$$

1. $\lambda(x{:}\ t_1).\ e$ has type $t_1 \rightarrow t_2$ in context Γ
2. $\lambda(x{:}\ t_1).\ e$ evaluates to $\langle\rho,\ (\lambda(x{:}\ t_1).\ e)\rangle$ in environment ρ
3. So if $\Gamma \vdash \rho$, then $\lambda(x{:}\ t1).\ e$ reduces to some value ($\langle\rho,\ (\lambda(x{:}\ t_1).\ e)\rangle$) of type $t_1 \rightarrow t_2$ in context Γ

Type safety proof sketch: applications

$$\text{T-App}\ \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2} \qquad \text{T-Cls}\ \frac{\Gamma' \vdash \rho \qquad x{:}\ t_1, \Gamma' \vdash e : t_2}{\Gamma \vdash \langle\rho,\ (\lambda(x{:}\ t_1).\ e)\rangle : t_1 \rightarrow t_2}$$

$$\text{E-App}\ \frac{\langle\rho,\ e_1\rangle \Downarrow \langle\rho',\ (\lambda(x{:}\ t_1).\ e'_1)\rangle \qquad \langle\rho,\ e_2\rangle \Downarrow v_2 \qquad \langle(x \mapsto v_2,\ \rho'),\ e'_1\rangle \Downarrow v}{\langle\rho,\ (e_1\ e_2)\rangle \Downarrow v}$$

1. e_1 reduces to some closure with type $t_1 \rightarrow t_2$ (induction hypothesis), with argument x, body e'_1, and some environment ρ' that's well-typed under some environment Γ'
2. e_2 reduces to some value v_2 with type t_1 (induction hypothesis)
3. e'_1 is well-typed under $(x \mapsto t_1,\ \Gamma')$, and reduces to some value v of type t_2
4. So $(e_1\ e_2)$ reduces to some value (v) of type t_2

QED (sort of)

This is just an high-level description of a proof; a full formal proof requires a bit more care and detail.

The takeaway is:

In a type-safe language, every well-typed expression reduces to a value of the same type.

A full type safety proof is often infeasible (or impossible) for a large practical language, but designers of statically-typed languages usually aim to get close to this goal.