Type Safety

Concepts of Programming Languages

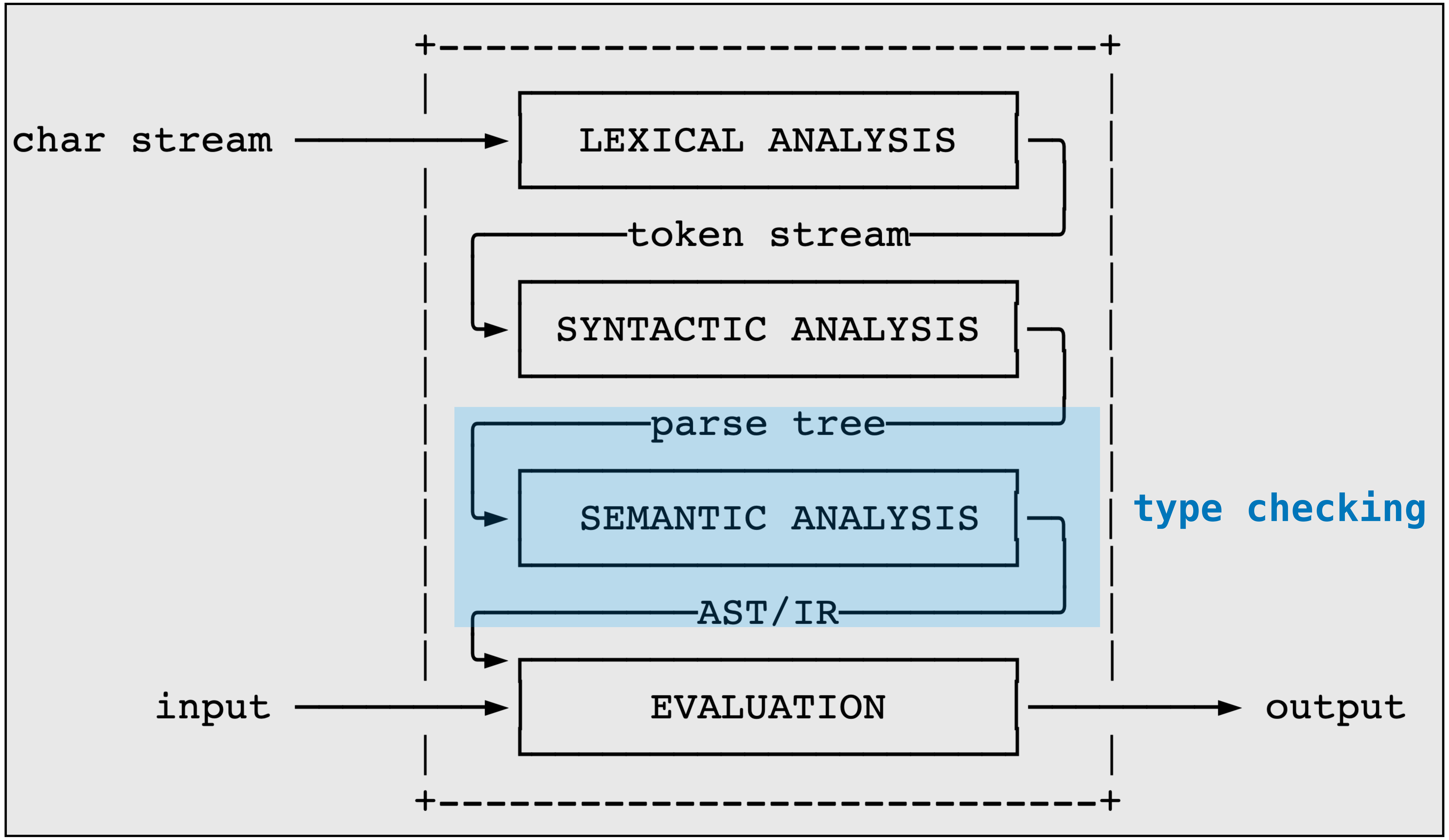
CAS CS 320

Outline

- Demo an **implementation** of the simply typed lambda calculus
- Discuss **induction** over derivations
- Show that STLC satisfies **progress** and **preservation**

Recap

Recall: The Picture



Recall: Type Checking/Inference

```
type_check : expr -> ty -> bool

type_of : expr -> ty option
```

Recall: Type Checking/Inference

```
type_check : expr -> ty -> bool

type_of : expr -> ty option
```

Type checking: determining whether an expression can be typed

Recall: Type Checking/Inference

```
type_check : expr -> ty -> bool
type_of : expr -> ty option
```

Type checking: determining whether an expression can be typed

Type inference: *synthesizing* a type for an expression

Recall: Type Checking/Inference

```
type_check : expr -> ty -> bool

type_of : expr -> ty option
```

Type checking: determining whether an expression can be typed

Type inference: *synthesizing* a type for an expression

Theoretically, these two problems can be very different. *For STLC, they are both easy*

Recall: Syntax (STLC)

$$e ::= \bullet \mid x \mid \lambda x^{\tau} . e \mid ee$$

$$\tau ::= \top \mid \tau \to \tau$$

$$x ::= \textit{variables}$$

`The syntax is the same as that of the lambda calculus except:`

» `we include a unit expression`

» `we have types, which annotate arguments`

Recall: Typing (STLC)

$$\frac{}{\Gamma \vdash \bullet : \top}\;\textbf{unit}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'}\;\textbf{abstraction}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\;\textbf{variable}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}\;\textbf{application}$$

These rules enforce that a function can only be applied if we *know* that it's a function

Recall: Semantics (STLC)

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}\ \texttt{leftEval} \qquad \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}\ \texttt{beta}$$

Recall: Semantics (STLC)

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}\ \texttt{leftEval} \qquad \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}\ \texttt{beta}$$

```
We can use any semantics (this is small-step CBN)
```

Recall: Semantics (STLC)

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}\ \textbf{leftEval} \qquad \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}\ \textbf{beta}$$

We can use any semantics (this is small-step CBN)

This is part of the point. Type-checking only determines *whether* we go on to evaluate the program

Recall: Semantics (STLC)

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}\ \texttt{leftEval} \qquad \frac{}{(\lambda x\,.\,e)e' \longrightarrow [e'/x]e}\ \texttt{beta}$$

We can use any semantics (this is small-step CBN)

This is part of the point. Type-checking only determines *whether* we go on to evaluate the program

It doesn't determine *how* we evaluate the program

Practice Problem

$$(\lambda x^{(\top \to \top) \to \top} . x(\lambda z^{\top} . x(wz)))y$$

Determine the smallest context such that the above expression is well-typed (also give its type)

$$\frac{}{\Gamma \vdash \bullet : \top}\ \textbf{unit}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textbf{variable}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'}\ \textbf{abstraction}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}\ \textbf{application}$$

Answer

$$(\lambda x^{(\mathsf{T}\to\mathsf{T})\to\mathsf{T}}\,.\,x(\lambda z^{\mathsf{T}}\,.\,x(wz)))y$$

$$\{w : \mathsf{T}\to(\mathsf{T}\to\mathsf{T}),\ y : (\mathsf{T}\to\mathsf{T})\to\mathsf{T}\}$$

$$\vdash \boxed{(\lambda x^{(\mathsf{T}\to\mathsf{T})\to\mathsf{T}}\,.\,\boxed{\underbrace{x}_{(\mathsf{T}\to\mathsf{T})\to\mathsf{T}}(\underbrace{\lambda z^{\mathsf{T}}\,.\,\underbrace{\underbrace{x}_{(\mathsf{T}\to\mathsf{T})\to\mathsf{T}}(w\underbrace{z}_{\mathsf{T}})}_{\mathsf{T}}}_{\mathsf{T}\to\mathsf{T}}))})}\,y : \mathsf{T}$$

demo

(STLC)

$$\frac{}{\Gamma \vdash () : \mathrm{unit}}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 : T}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \boxed{\Gamma, x : \tau} \vdash e_2 : \tau'}{\Gamma \vdash \text{let } x : \tau = e_1 \text{ in } e_2 : \tau'}$$

$$\frac{\Gamma, f:\tau_a \to \tau_o, x:\tau_a \vdash e_1 : \tau_o \qquad \Gamma, f:\tau_a \to \tau_o \vdash e_2 : \tau}{\Gamma \vdash \text{let rec } f\ (x:\tau_a) : \tau_o = e_1 \text{ in } e_2 : \tau}$$

Type Safety

How do we know if we've defined a "good" programming language?

Type Safety

Type Safety

if e is terminating

Theorem. If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

Type Safety

> **Theorem.** If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

With small-step semantics, we can give a finer-grained analysis:

Type Safety

> **Theorem.** If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

With small-step semantics, we can give a finer-grained analysis:

no stuck term

> **Theorem.** If $\cdot \vdash e : \tau$, then
>
> » *(progress)* either e is a value or there is an e' such that $e \longrightarrow e'$
>
> » *(preservation)* If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$

Type Safety

> **Theorem.** If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

With small-step semantics, we can give a finer-grained analysis:

> **Theorem.** If $\cdot \vdash e : \tau$, then
>
> » *(progress)* either e is a value or there is an e' such that $e \longrightarrow e'$
>
> » *(preservation)* If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$

These results are *fundamental*. They tell us that our PL is well-behaved (it's a "good" PL)

Induction over Derivations

The Key Idea

The Key Idea

$$\frac{\dfrac{}{\{\} \vdash \mathtt{2} : \mathtt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\,(\text{var}) \qquad \dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\,(\text{var})}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} + \mathtt{y} : \mathtt{int}}\,(\text{intAdd})}{\{\} \vdash \mathtt{let\ y\ =\ 2\ in\ y} + \mathtt{y} : \mathtt{int}}\,(\text{let})$$

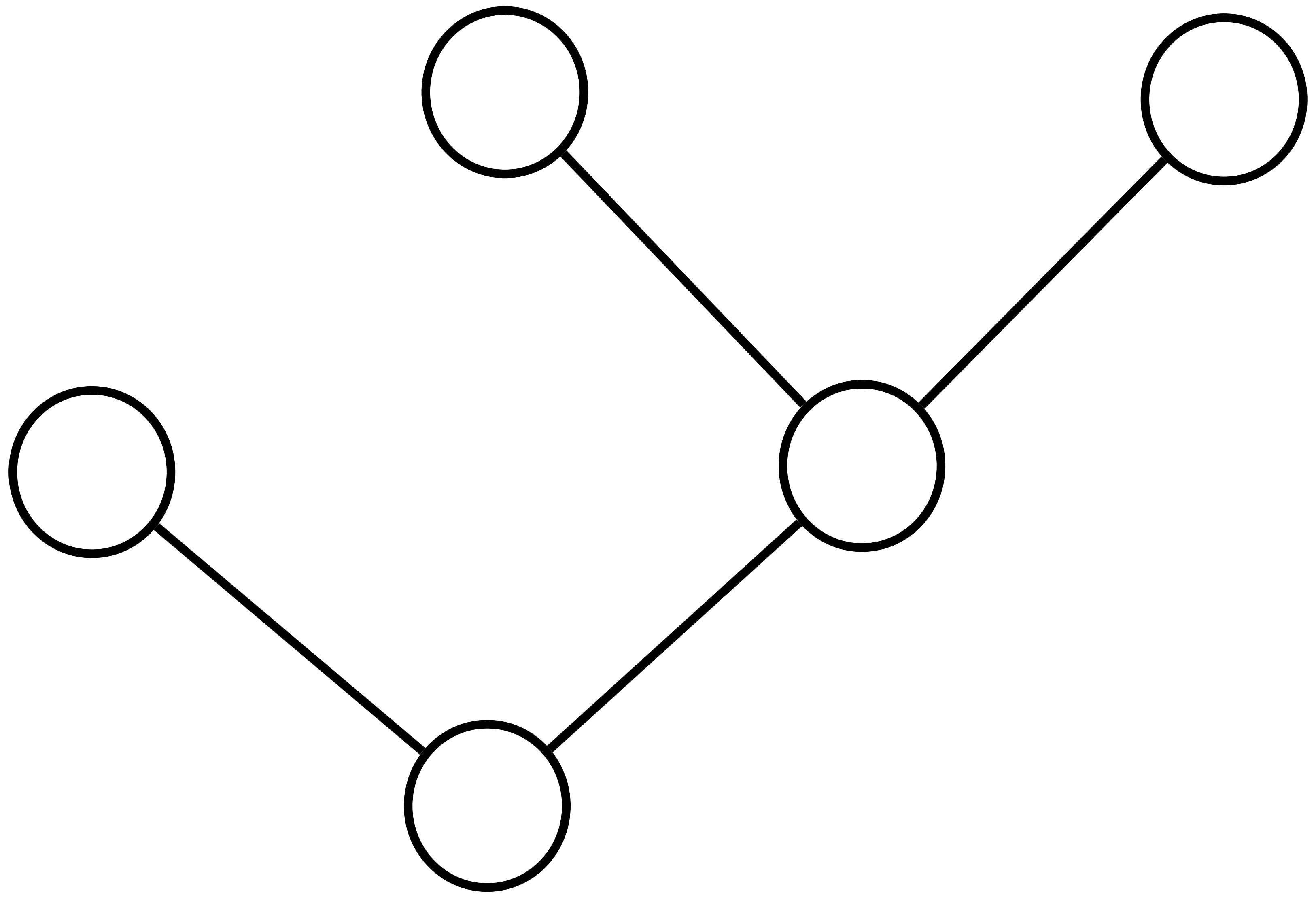
Derivations are *trees*

The Key Idea

$$\dfrac{\dfrac{}{\{\} \vdash \texttt{2} : \texttt{int}}\ (\mathsf{intLit}) \qquad \dfrac{\dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\ (\mathsf{var}) \qquad \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\ (\mathsf{var})}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} + \texttt{y} : \texttt{int}}\ (\mathsf{intAdd})}{\{\} \vdash \texttt{let y = 2 in } \texttt{y} + \texttt{y} : \texttt{int}}\ (\mathsf{let})$$

Derivations are *trees*

We can prove things about trees using induction



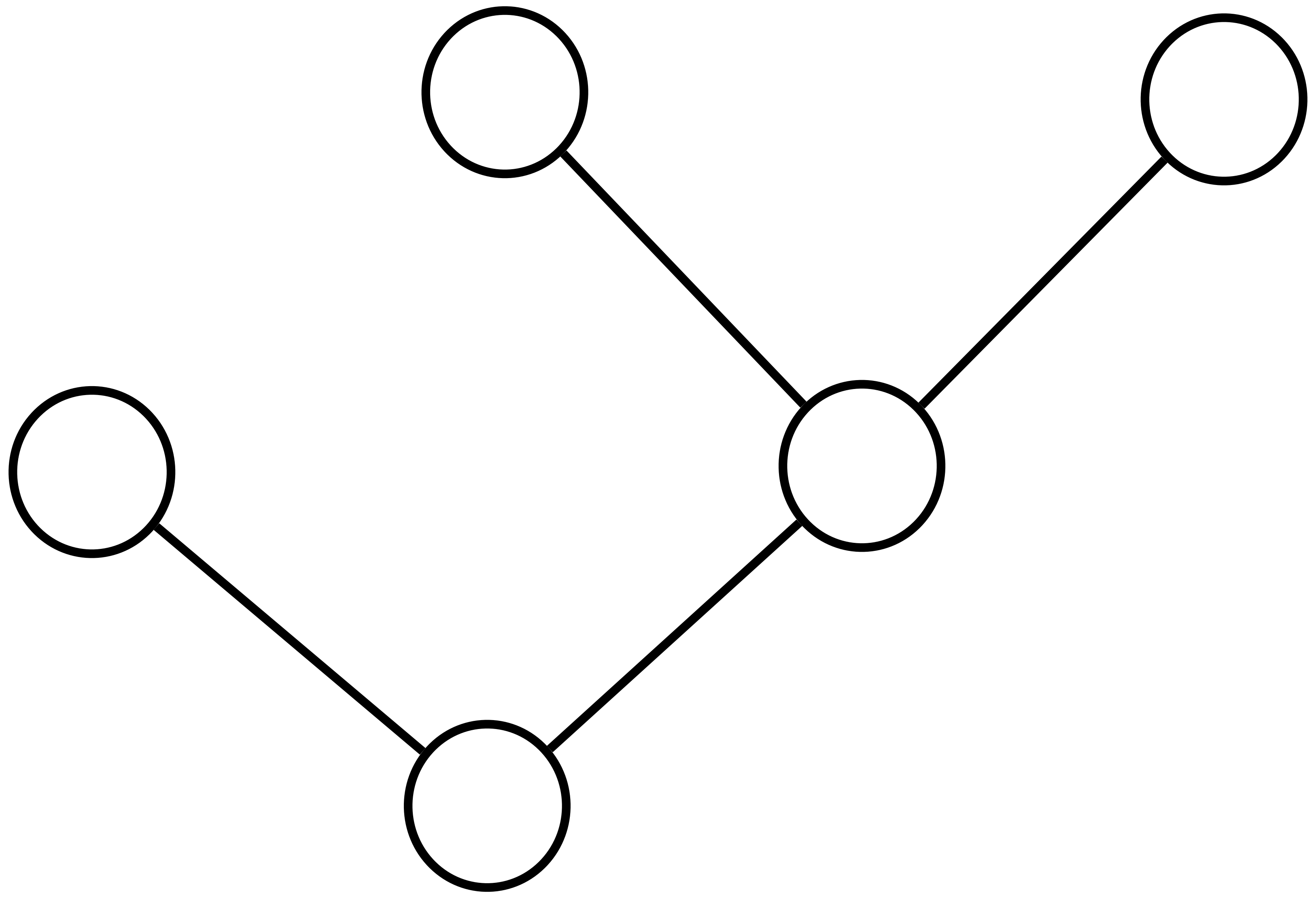
The Key Idea

$$\dfrac{\dfrac{}{\{\} \vdash \texttt{2} : \texttt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var}) \qquad \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var})}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} + \texttt{y} : \texttt{int}}\,(\text{intAdd})}{\{\} \vdash \texttt{let y = 2 in y} + \texttt{y} : \texttt{int}}\,(\text{let})$$

Derivations are *trees*

We can prove things about trees using induction

We can prove things about *derivable judgments* using induction



The Key Idea

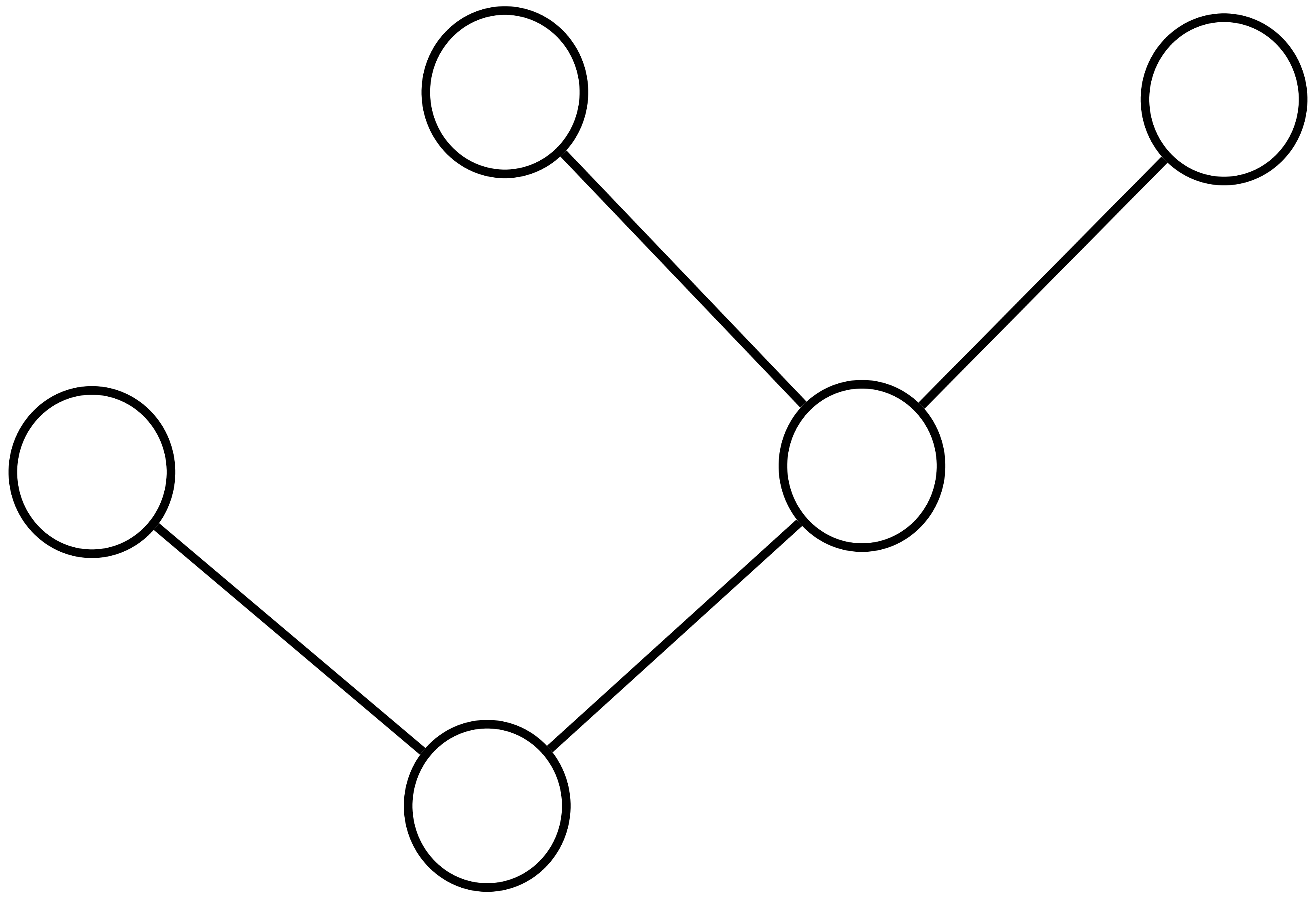
$$\dfrac{\dfrac{}{\{\} \vdash \texttt{2} : \texttt{int}}\,(\textsf{intLit}) \qquad \dfrac{\dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\textsf{var}) \qquad \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\textsf{var})}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} + \texttt{y} : \texttt{int}}\,(\textsf{intAdd})}{\{\} \vdash \texttt{let y = 2 in y} + \texttt{y} : \texttt{int}}\,(\textsf{let})$$

Derivations are *trees*

We can prove things about trees using induction

We can prove things about *derivable judgments* using induction

Important: Every derivable judgment corresponds to a derivation



Warm-up: Binary Trees

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Warm-up: Binary Trees

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Let $\text{size}(T)$ denote the number of Nodes and let $\text{height}(T)$ denote the length of the *longest* path from the root to any Node

Warm-up: Binary Trees

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Let $\text{size}(T)$ denote the number of Nodes and let $\text{height}(T)$ denote the length of the *longest* path from the root to any Node

Theorem. $\text{size}(T) \leq 2^{\text{height}(T)} - 1$ for any tree T

Warm-up: Binary Trees

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Let $\text{size}(T)$ denote the number of Nodes and let $\text{height}(T)$ denote the length of the *longest* path from the root to any Node

> **Theorem.** $\text{size}(T) \leq 2^{\text{height}(T)} - 1$ for any tree T

Proof. By induction on the structure of trees

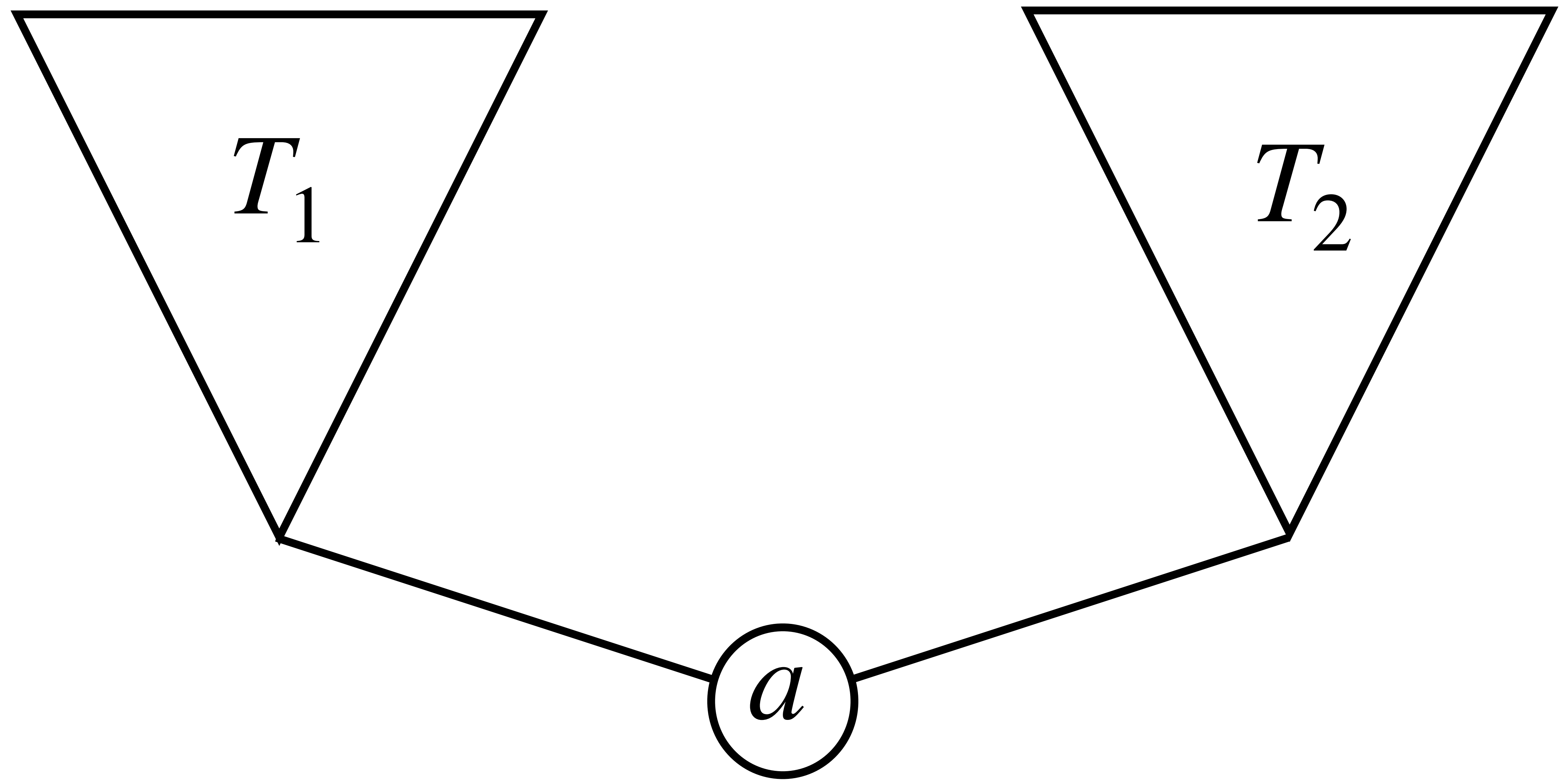
Base Case: Empty

$\text{size}(T) = 0$

$\text{height}(T) = 0$

$0 \leq 2^0 - 1 = 0$

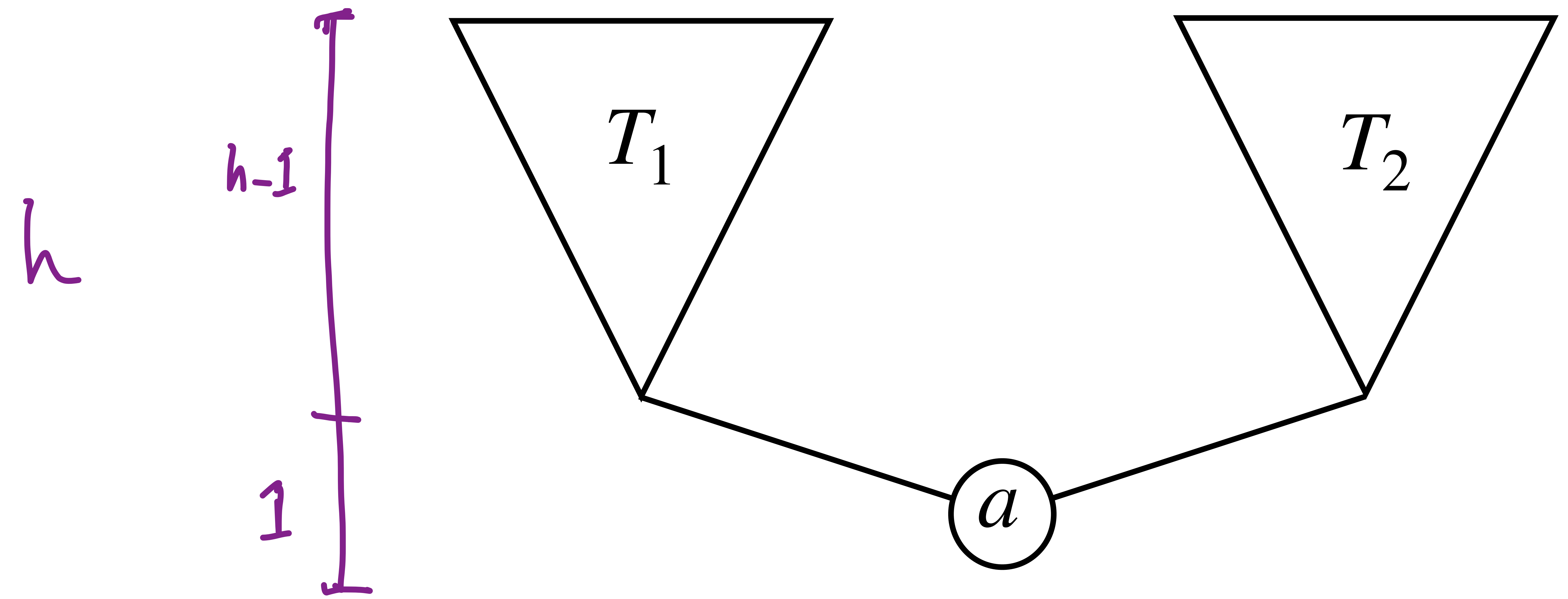
Inductive Hypothesis



If T is of the form `Node` (T_1, a, T_2) then $\text{size}(T_1) \leq 2^{\text{height}(T_1)} - 1$ and $\text{size}(T_2) \leq 2^{\text{height}(T_2)} - 1$

That is, we get to assume that what we want holds of our subtrees

Inductive Step: Nodes



$$\mathrm{size}(T) = \mathrm{size}(T_1) + \mathrm{size}(T_2) + 1$$
$$\leq 2^{h(T_1)} - 1 + 2^{h(T_2)} - 1 + 1$$
$$\leq \boxed{2^{h(T)-1}} - 1 + \boxed{2^{h(T)-1}} \cancel{-1} + \cancel{1} = 2^{h(T)} - 1$$

Another Warm-up: Well-Scopedness

Another Warm-up: Well-Scopedness

An expression e is **well-scoped** with respect to a context Γ if $x \in FV(e)$ implies x appears in Γ

Another Warm-up: Well-Scopedness

An expression e is **well–scoped** with respect to a context Γ if $x \in FV(e)$ implies x appears in Γ

> Theorem. If e is well–typed in Γ, then e is well–scoped

Another Warm-up: Well-Scopedness

An expression e is **well-scoped** with respect to a context Γ if $x \in FV(e)$ implies x appears in Γ

Theorem. If e is well-typed in Γ, then e is well-scoped

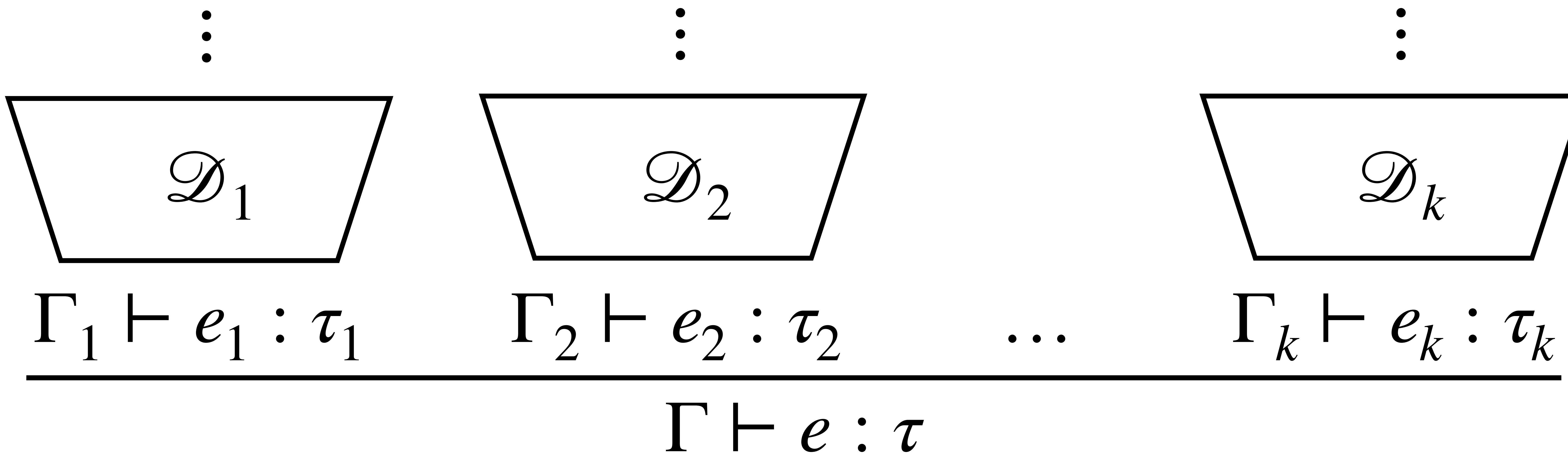
Proof. By induction on derivations

Base Case: Axioms

$$\frac{}{\Gamma \vdash \bullet : \top}\,\texttt{unit} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\,\texttt{variable}$$

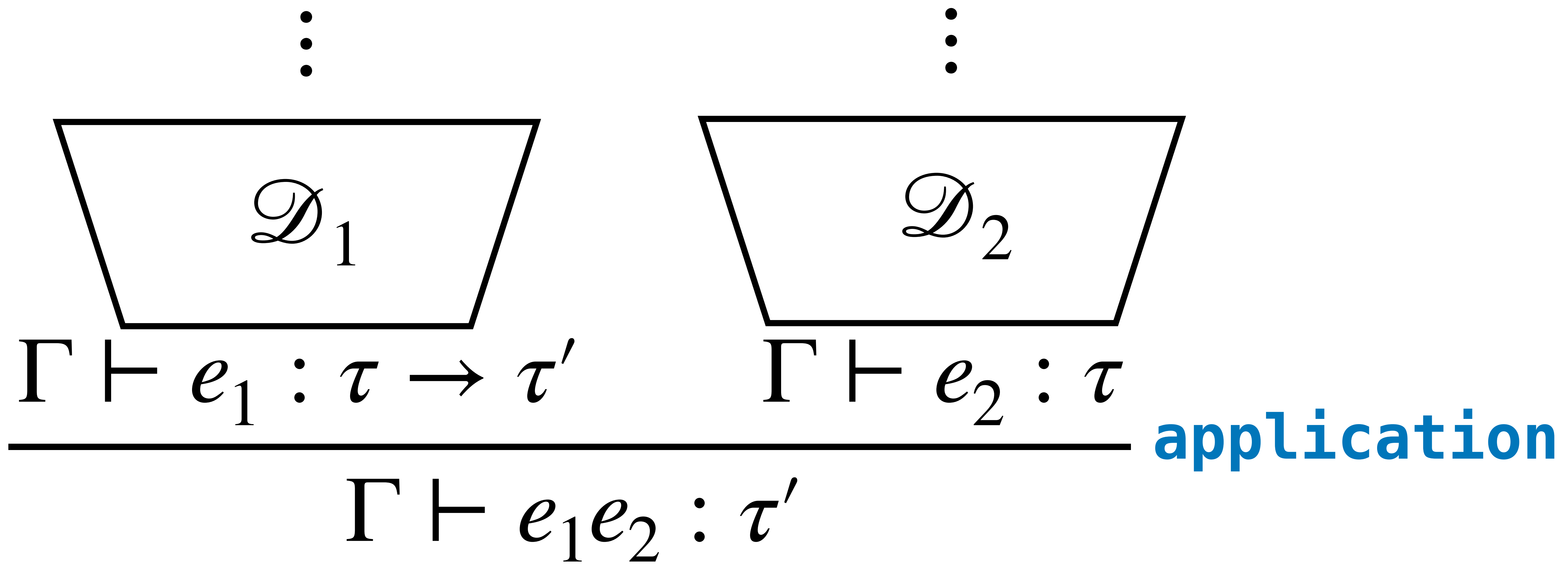
We need to show that expressions typed using just axioms satisfy well-scopedness

Inductive Hypothesis


$\mathscr{D}_1$
$\mathscr{D}_2$
$\mathscr{D}_k$
$\Gamma_1 \vdash e_1 : \tau_1$
$\Gamma_2 \vdash e_2 : \tau_2$
$\dots$
$\Gamma_k \vdash e_k : \tau_k$
$\Gamma \vdash e : \tau$

If $e_1, \dots, e_k$ are well-scoped (because they are typeable in the each of their contexts)

Inductive Step 1: Application



$$\frac{\begin{array}{c}\vdots\\ \mathscr{D}_1\\ \Gamma \vdash e_1 : \tau \to \tau'\end{array} \qquad \begin{array}{c}\vdots\\ \mathscr{D}_2\\ \Gamma \vdash e_2 : \tau\end{array}}{\Gamma \vdash e_1 e_2 : \tau'}\ \texttt{application}$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$$

What if the last rule I applied was application?

Inductive Step 2: Abstraction

$$\dfrac{\begin{array}{c}\vdots \\ \mathscr{D} \\ \Gamma, x : \tau \vdash e : \tau'\end{array}}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'}\ \textbf{abstraction}$$

Challenge exercise

What if the last rule I applied was abstraction?

Progress and Preservation

Recall: Type Safety

Recall: Type Safety

> Theorem. If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

Recall: Type Safety

> Theorem. If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

With small-step semantics, we can give a finer-grained analysis:

Recall: Type Safety

> Theorem. If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

With small-step semantics, we can give a finer-grained analysis:

> Theorem. If $\cdot \vdash e : \tau$, then
>
> » *(progress)* either e is a value or there is an e' such that $e \longrightarrow e'$
>
> » *(preservation)* If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$

Recall: Type Safety

> Theorem. If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

With small-step semantics, we can give a finer-grained analysis:

> Theorem. If $\cdot \vdash e : \tau$, then
>
> » *(progress)* either e is a value or there is an e' such that $e \longrightarrow e'$
>
> » *(preservation)* If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$

These results are *fundamental*. They tell us that our PL is well-behaved (it's a "good" PL)

Recall: Type Safety

> Theorem. If $\cdot \vdash e : \tau$ then there is a value v such that $\langle \varnothing, e \rangle \Downarrow v$ and $\cdot \vdash v : \tau$

With small-step semantics, we can give a finer-grained analysis:

goal for today

> Theorem. If $\cdot \vdash e : \tau$, then
>
> » *(progress)* either e is a value or there is an e' such that $e \longrightarrow e'$
>
> » *(preservation)* If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$

These results are *fundamental*. They tell us that our PL is well-behaved (it's a "good" PL)

Disclaimer: We're gonna hand-wave liberally

Recall: STLC

$$e ::= \bullet \mid x \mid \lambda x^{\tau}.e \mid ee$$

$$\tau ::= \top \mid \tau \to \tau$$

$$x ::= \textit{variables}$$

Typing

$$\frac{}{\Gamma \vdash \bullet : \top}\ \texttt{unit}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau}.e : \tau \to \tau'}\ \texttt{abstraction}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \texttt{variable}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}\ \texttt{application}$$

Semantics

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}\ \texttt{leftEval}$$

$$\frac{}{(\lambda x.e)e' \longrightarrow [e'/x]e}\ \texttt{beta}$$

Progress (STLC)

> **Theorem.** If e is well-typed ($\cdot \vdash e : \tau$ for some type τ), then e is a value, or there is an expression e' such that $e \longrightarrow e'$

Proof. By induction over <u>derivations</u>

Base Case: Axioms

$$\frac{}{\cdot \vdash \bullet : \top}\ \textbf{unit} \qquad \frac{(x : \tau) \in \varnothing}{\cdot \vdash x : \tau}\ \textbf{variable}$$

We need to show that expressions typed using just axioms yield non-stuck terms

Inductive Step 1: Application

$$\frac{\cdot \vdash e_1 : \tau \rightarrow \tau' \qquad \cdot \vdash e_2 : \tau}{\cdot \vdash e_1 e_2 : \tau'}\ \texttt{application}$$

*What do we know given that e_1 is either a **value** or **reducible**?*

Inductive Step 2: Abstraction

$$\frac{\{x : \tau\} \vdash e : \tau'}{\cdot \vdash \lambda x^{\tau} . e : \tau \to \tau'}\ \textbf{abstraction}$$

Our expression already a value if the last rule we applied was abstraction!

Preservation (STLC)

> Theorem. If e has type τ in Γ (i.e., $\Gamma \vdash e : \tau$ is derivable) and $e \longrightarrow e'$ then so is e' (i.e., $\Gamma \vdash e' : \tau$ is derivable)

Proof. By induction over derivations

This one is much tricker...

Base Case: Axioms

$$\frac{}{\Gamma \vdash \bullet : \top}\,\textbf{unit} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\,\textbf{variable}$$

Expressions typed using just axioms cannot be reduced (nothing to do here)

Inductive Step 1: Abstraction

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x^{\tau} . e : \tau \to \tau'}\ \texttt{abstraction}$$

Expressions derived using abstraction as the last rule is already a value (nothing to do here)

Inductive Step 2: Application

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}\ \textbf{application}$$

This is where the work comes in...

<u>The trick:</u> We do induction (inside our current induction) on the structure of *semantic* derivations!

What possible ways can $e_1 e_2$ *be reduced?*

Inductive Step 2.1: leftEval

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2}\ \texttt{leftEval} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}\ \texttt{application}$$

What if our last rule was an application ***and*** e_1e_2 *is reducible by leftEval?*

Inductive Step 2.1: leftEval

$$\frac{}{(\lambda x\,.\,e)e_2 \longrightarrow [e_2/x]e}\,\textbf{beta}$$

$$\frac{\Gamma \vdash (\lambda x\,.\,e) : \tau \rightarrow \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\lambda x\,.\,e)e_2 : \tau'}\,\textbf{application}$$

What if our last rule was an application ***and*** e_1e_2 *is reducible by beta?*

Substitution Lemma

> <u>Lemma.</u> If $\Gamma \vdash e_2 : \tau_2$ and $\Gamma, x : \tau_2 \vdash e : \tau$ then
>
> $$\Gamma \vdash [e_2/x]e : \tau$$

That is, if e is well-typed in a context with (x, τ) then we can substitute x with anything of type τ and it's still the same type

(we can prove this by, you guessed it, induction on derivations)

The Point

```
let rec eval env e =
  match e with
  | Var x -> Env.find x env
  ...
```

Progress and preservation tell us that **terms never get stuck during evaluation**

*This is **HUGE.** I can't emphasize this enough*

Our type system ensures we only evaluate programs that make sense!

Summary

Progress and **preservation** are fundamental features of good programming languages

We can prove things about well-typed expressions by performing **induction** over derivations