Type Safety

Concepts of Programming Languages
Lecture 19

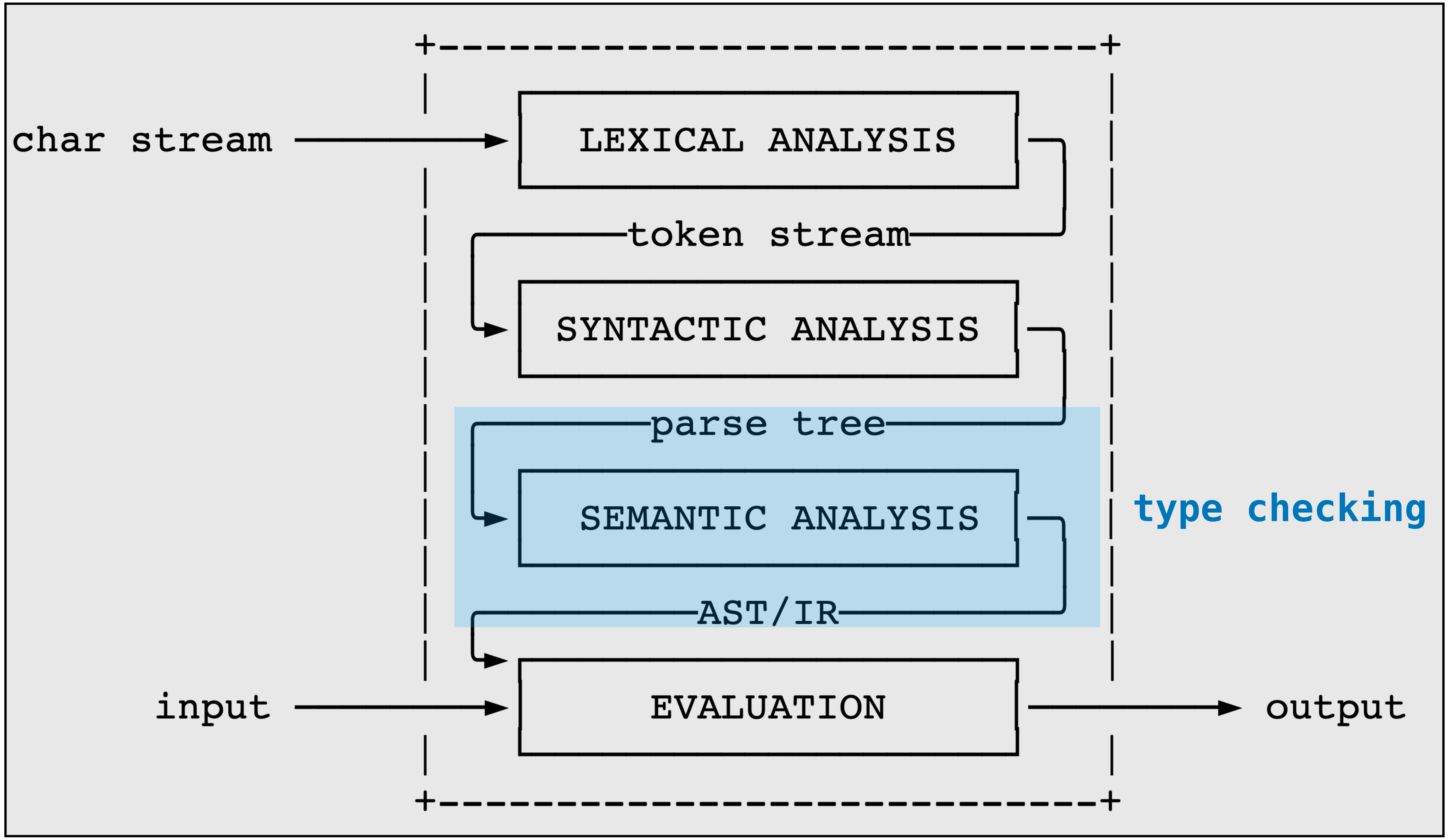
CAS CS 320

Outline

- Demo an **implementation** of the simply typed lambda calculus and desugaring
- 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
```

<u>Type checking:</u> determining whether an expression can be typed

Recall: Type Checking/Inference

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

type_of : expr -> ty option
```

<u>Type checking:</u> determining whether an expression can be typed

<u>Type inference:</u> *synthesizing* a type for an expression

Recall: Type Checking/Inference

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

type_of : expr -> ty option
```

<u>Type checking:</u> determining whether an expression can be typed

<u>Type inference:</u> *synthesizing* a type for an expression

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

Recall: Syntax (STLC)

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

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

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

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)

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

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

Recall: Semantics (STLC)

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

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

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

Recall: Semantics (STLC)

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

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

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

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 typeable (also give its type)

Answer

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

demo

(STLC)

Induction over Derivations

The Key Idea

The Key Idea

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

Derivations are *trees*

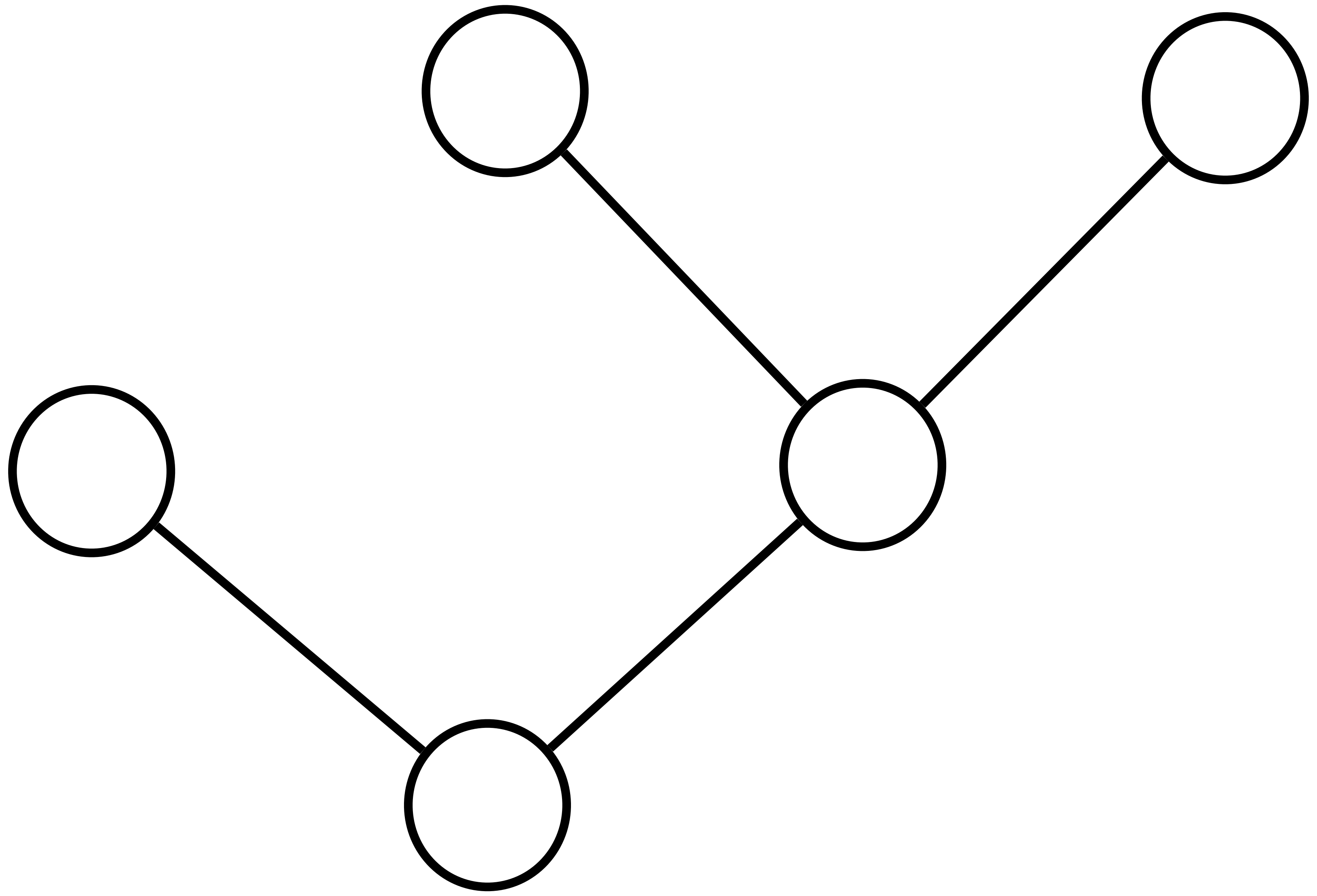
The Key Idea

$$\dfrac{}{\{\} \vdash \mathtt{2} : \mathtt{int}}\,(\mathsf{intLit}) \quad \dfrac{\dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\,(\mathsf{var}) \quad \dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\,(\mathsf{var})}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} + \mathtt{y} : \mathtt{int}}\,(\mathsf{intAdd})$$

$$\dfrac{\{\} \vdash \mathtt{2} : \mathtt{int} \qquad \{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} + \mathtt{y} : \mathtt{int}}{\{\} \vdash \mathtt{let\ y\ =\ 2\ in\ y} + \mathtt{y} : \mathtt{int}}\,(\mathsf{let})$$

Derivations are *trees*

We can prove things about trees using induction



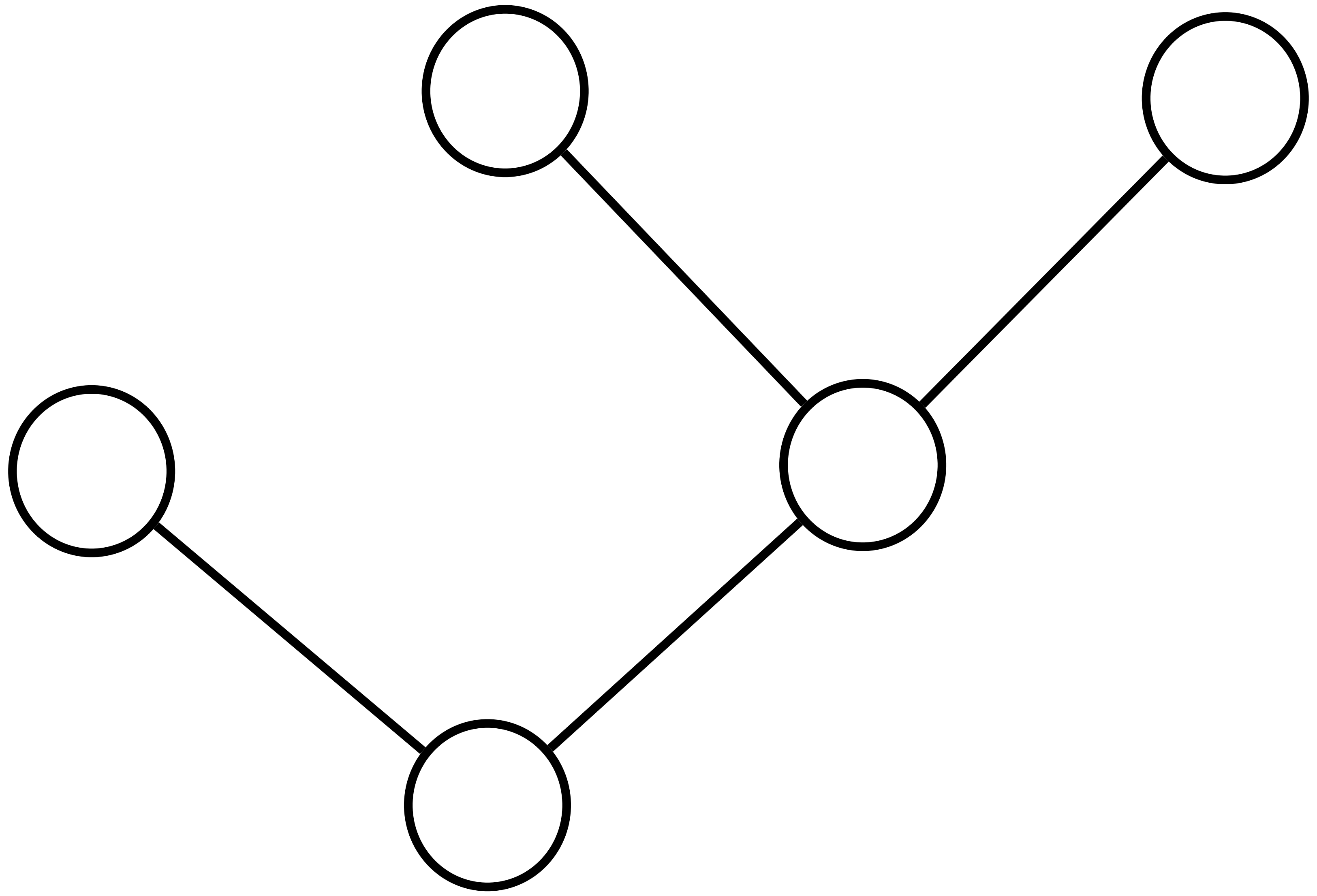
The Key Idea

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

Derivations are *trees*

We can prove things about trees using induction

We can prove things about *derivable judgments* using induction



The Key Idea

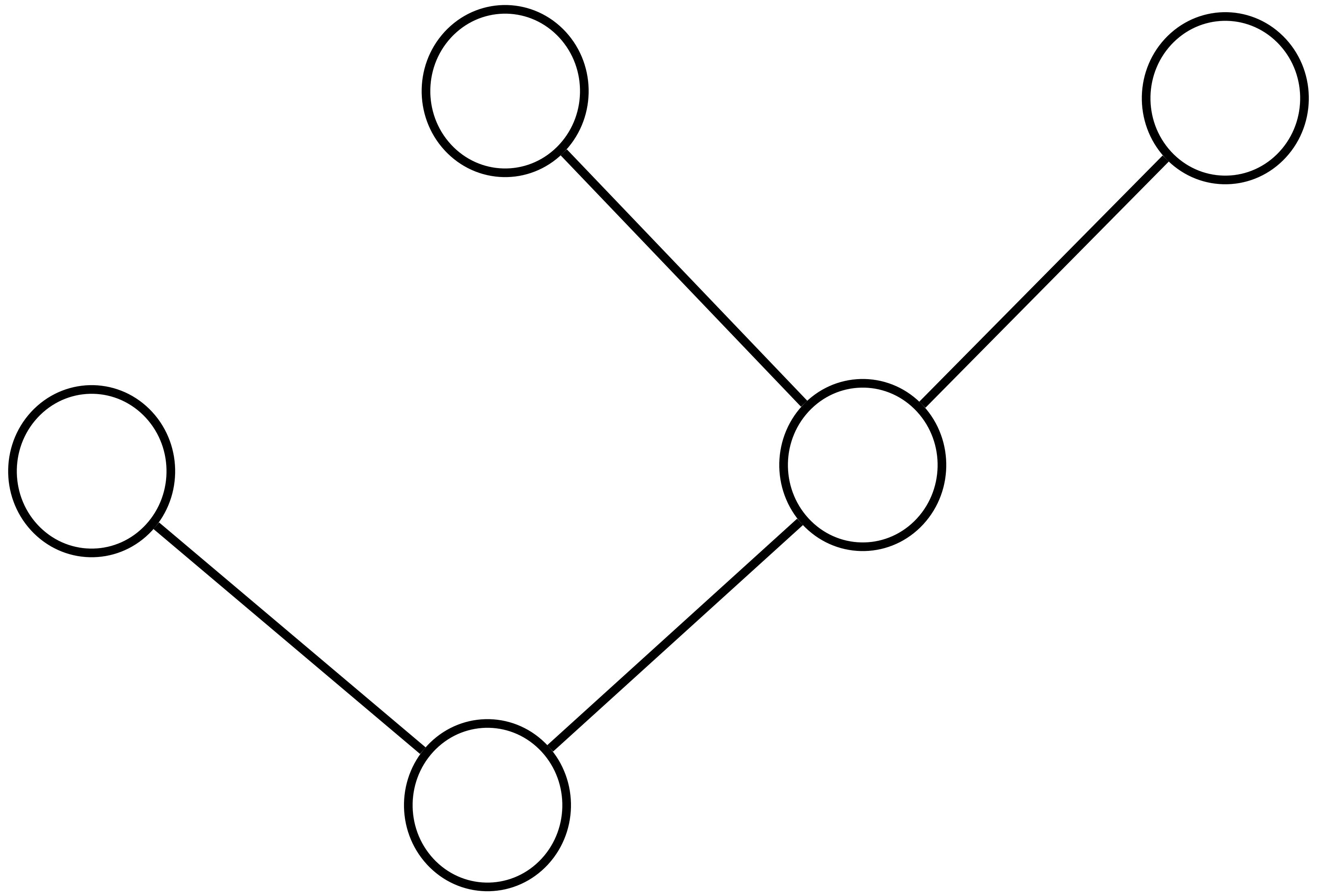
$$\dfrac{\dfrac{}{\{\} \vdash 2 : \texttt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var}) \qquad \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var})}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y}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

Derivations are *trees*

We can prove things about trees using induction

We can prove things about *derivable judgments* using induction

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 $\mathsf{size}(T)$ denote the number of Nodes and let $\mathsf{height}(T)$ denote the length of the *longest* path from the root to Empty

Warm-up: Binary Trees

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

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

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

Warm-up: Binary Trees

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

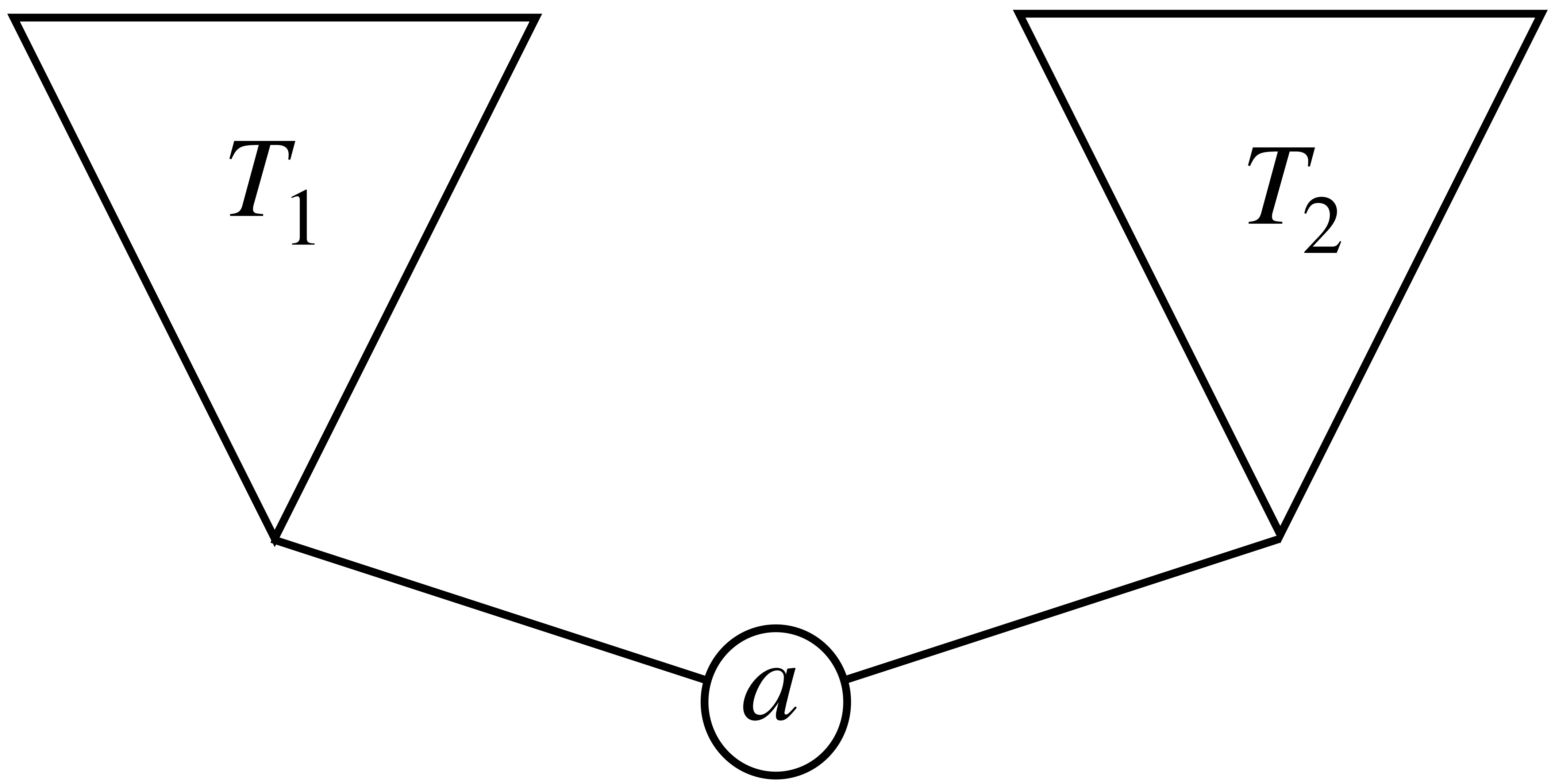
Let $\text{size}(T)$ denote the number of `Node`s and let $\text{height}(T)$ denote the length of the *longest* path from the root to `Empty`

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

Proof. By induction on the <u>structure of trees</u>

Base Case: Empty

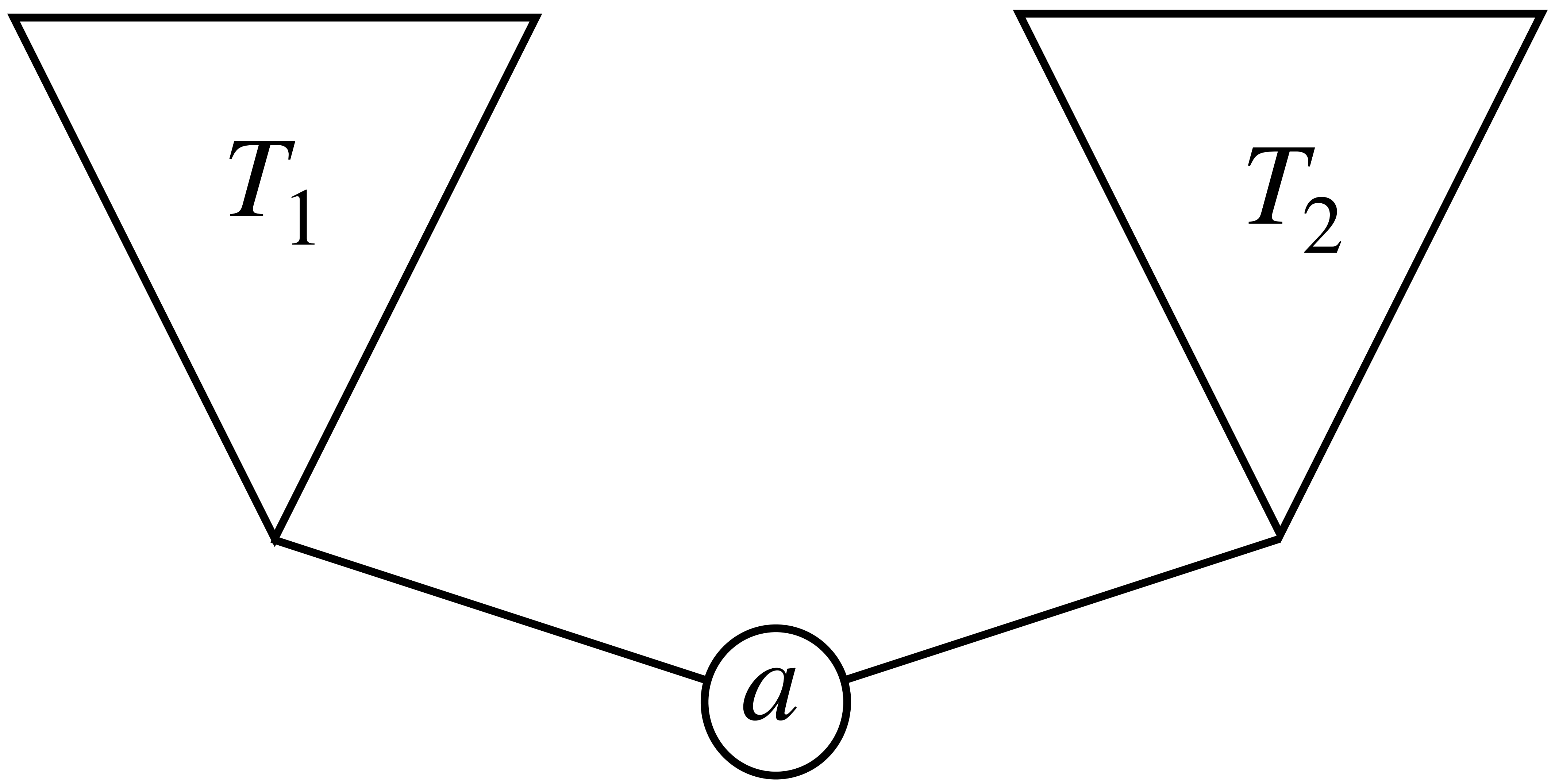
Inductive Hypothesis



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

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

Inductive Step: Nodes



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

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

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

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

Inductive Hypothesis

$$\dfrac{\begin{array}{c}\vdots\\ \mathcal{D}_1\\ \Gamma_1 \vdash e_1 : \tau_1\end{array} \qquad \begin{array}{c}\vdots\\ \mathcal{D}_2\\ \Gamma_2 \vdash e_2 : \tau_2\end{array} \qquad \dots \qquad \begin{array}{c}\vdots\\ \mathcal{D}_k\\ \Gamma_k \vdash e_k : \tau_k\end{array}}{\Gamma \vdash e : \tau}$$

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

Inductive Step 1: Application

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

`What if the last rule I applied was application?`

Inductive Step 2: Abstraction

$$\vdots$$

$$\mathscr{D}$$

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

What if the last rule I applied was abstraction?

Progress and Preservation

Recall: Type Safety

Recall: Type Safety

<u>Theorem.</u> 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

> <u>Theorem.</u> 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:

> <u>Theorem.</u> 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 derivations

Base Case: Axioms

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

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

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

Inductive Step 1: Application

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

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

Inductive Step 2: Abstraction

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

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

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

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

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

Inductive Step 1: Abstraction

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

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

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

What if our last rule was an application ***and*** $e_1 e_2$ *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** $e_1 e_2$ is reducible by beta?*

Substitution Lemma

> Lemma. 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