Outline

- We will write type systems for multiple languages
Outline

- We will write type systems for multiple languages
- We will formally see how to define soundness
Outline

- We will write type systems for multiple languages
- We will formally see how to define soundness
- We will learn how to prove soundness of a type system
The let language

Recall from last time the following small language (let language):

\[ S \rightarrow \text{integer} \mid \text{string} \mid \text{identifier} \mid S_1 + S_2 \mid S_1 :: S_2 \mid \text{let } id : \tau = S_1 \text{ in } S_2 \]

\[ \tau \rightarrow \text{Int} \mid \text{String} \]
The let language

- Recall from last time the following small language (let language):

\[
S \rightarrow \text{integer} \mid \text{string} \mid \text{identifier} \\
\mid S_1 + S_2 \mid S_1 :: S_2 \\
\mid \text{let } id : \tau = S_1 \text{ in } S_2
\]

\[
\tau \rightarrow \text{Int} \mid \text{String}
\]

- Here are again its operational semantics:

\[
\begin{array}{c}
\text{integer } i \\
\text{string } s \\
\text{identifier } id
\end{array}
\quad
\begin{array}{c}
E \vdash i : i \\
E \vdash s : s \\
E \vdash id : E(id)
\end{array}
\quad
\begin{array}{c}
E \vdash S_1 : i_1 \\
E \vdash S_2 : i_2
\end{array}
\quad
\begin{array}{c}
E \vdash S_1 : i_1 + i_2
\end{array}
\]

\[
\begin{array}{c}
E \vdash S_1 : s_1 \\
E \vdash S_2 : s_2
\end{array}
\quad
\begin{array}{c}
E \vdash S_1 : e_1 \\
E \vdash S_2 : e_2
\end{array}
\quad
\begin{array}{c}
E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2
\end{array}
\]

\[
E \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)
\]
We also saw last time how we can write **typing rules** that compute the type of an expression.

- **integer** $i$
  \[
  \Gamma \vdash i : \text{Int} \n  \]
- **string** $s$
  \[
  \Gamma \vdash s : \text{String} \n  \]
- **identifier** $id$
  \[
  \Gamma \vdash id : \Gamma(id) \n  \]

- $\Gamma \vdash S_1 : \text{Int}$
  \[
  \Gamma \vdash S_2 : \text{Int} \n  \]
  \[
  \Gamma \vdash S_1 + S_2 : \text{Int} \n  \]
- $\Gamma \vdash S_1 : \text{String}$
  \[
  \Gamma \vdash S_1 :: S_2 : \text{String} \n  \]

- $\Gamma \vdash S_1 : \tau_1$
  \[
  \tau = \tau_1 \n  \]
  \[
  \Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \n  \]
  \[
  \Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3 \n  \]
Correspondence between Concrete and Abstract Semantics

- Observe that there is a strong relationship between the operational semantics (concrete semantics) and the typing rules (abstract semantics)
Correspondence between Concrete and Abstract Semantics

- Observe that there is a strong relationship between the operational semantics (concrete semantics) and the typing rules (abstract semantics).
  - The concrete **environment** $E$ corresponds to the abstract type environment $\Gamma$. 

Correspondence between Concrete and Abstract Semantics

- Observe that there is a strong relationship between the operational semantics (concrete semantics) and the typing rules (abstract semantics)
  - The concrete \textit{environment} $E$ corresponds to the abstract type environment $\Gamma$
  - The structure of the abstract and concrete rules are analogous
Correspondence between Concrete and Abstract Semantics

- Observe that there is a strong relationship between the operational semantics (concrete semantics) and the typing rules (abstract semantics)
  - The concrete environment \( E \) corresponds to the abstract type environment \( \Gamma \)

- The structure of the abstract and concrete rules are analogous

- **Key Difference**: Concrete semantics compute a particular value, while abstract semantics compute a type
Some Notation

- We write $\gamma(\tau)$ for the concretization of the abstract value $\tau$. We call $\gamma$ the concretization function.
Some Notation

- We write $\gamma(\tau)$ for the concretization of the abstract value $\tau$. We call $\gamma$ the concretization function.

- Example: $\gamma(\text{Int}) = \ldots, -1, 0, 1, 2, 3, \ldots$

- We write $\alpha(v)$ for the abstraction of the concrete value $v$. We call $\alpha$ the abstraction function.

- Example: $\alpha(42) = \text{Int}$

- Definition: An abstraction is a Galois Connection if $\alpha(\gamma(\tau)) = \tau$ for all abstract values $\tau$. 

- Question: Is our abstract domain of types a Galois connection? Yes, $\alpha(\gamma(\text{Int})) = \text{Int}$ and $\alpha(\gamma(\text{String})) = \text{String}$.
Some Notation

- We write $\gamma(\tau)$ for the concretization of the abstract value $\tau$. We call $\gamma$ the concretization function.

- Example: $\gamma(\text{Int}) = \{ \ldots, -1, 0, 1, 2, 3, \ldots \}$
Some Notation

- We write $\gamma(\tau)$ for the concretization of the abstract value $\tau$. We call $\gamma$ the concretization function.

- Example: $\gamma(\text{Int}) = \{\ldots, -1, 0, 1, 2, 3, \ldots\}$

- We write $\alpha(v)$ for the abstraction of the concrete value $v$. We call $\alpha$ the abstraction function.
Some Notation

- We write $\gamma(\tau)$ for the **concretization** of the abstract value $\tau$. We call $\gamma$ the concretization function.

- Example: $\gamma(\text{Int}) = \{\ldots, -1, 0, 1, 2, 3, \ldots\}$

- We write $\alpha(v)$ for the **abstraction** of the concrete value $v$. We call $\alpha$ the abstraction function.

- Example: $\alpha(42) = \ldots$
Some Notation

- We write $\gamma(\tau)$ for the concretization of the abstract value $\tau$. We call $\gamma$ the concretization function.

- Example: $\gamma(\text{Int}) = \{\ldots, -1, 0, 1, 2, 3, \ldots\}$

- We write $\alpha(v)$ for the abstraction of the concrete value $v$. We call $\alpha$ the abstraction function.

- Example: $\alpha(42) = \text{Int}$
Some Notation

- We write $\gamma(\tau)$ for the concretization of the abstract value $\tau$. We call $\gamma$ the concretization function.

- Example: $\gamma(\text{Int}) = \{\ldots, -1, 0, 1, 2, 3, \ldots\}$

- We write $\alpha(v)$ for the abstraction of the concrete value $v$. We call $\alpha$ the abstraction function.

- Example: $\alpha(42) = \text{Int}$

- Definition: An abstraction is a Galois Connection if $\alpha(\gamma(\tau)) = \tau$ for all abstract values $\tau$. 
Some Notation

- We write $\gamma(\tau)$ for the concretization of the abstract value $\tau$. We call $\gamma$ the concretization function.

- Example: $\gamma(\text{Int}) = \{-1, 0, 1, 2, 3, \ldots\}$

- We write $\alpha(v)$ for the abstraction of the concrete value $v$. We call $\alpha$ the abstraction function.

- Example: $\alpha(42) = \text{Int}$

- Definition: An abstraction is a Galois Connection if $\alpha(\gamma(\tau)) = \tau$ for all abstract values $\tau$.

- Question: Is our abstract domain of types a Galois connection?
Some Notation

- We write $\gamma(\tau)$ for the **concretization** of the abstract value $\tau$. We call $\gamma$ the concretization function.

- **Example:** $\gamma(\text{Int}) = \{\ldots, -1, 0, 1, 2, 3, \ldots\}$

- We write $\alpha(v)$ for the **abstraction** of the concrete value $v$. We call $\alpha$ the abstraction function.

- **Example:** $\alpha(42) = \text{Int}$

- **Definition:** An abstraction is a **Galois Connection** if $\alpha(\gamma(\tau)) = \tau$ for all abstract values $\tau$.

- **Question:** Is our abstract domain of types a Galois connection? Yes, $\alpha(\gamma(\text{Int})) = \text{Int}$ and $\alpha(\gamma(\text{String})) = \text{String}$.
Galois Connection

- Galois connection means that if we want to relate a concrete value $v$ and abstract value $\tau$, the following are equivalent:
Galois Connection

- Galois connection means that if we want to relate a concrete value \( v \) and abstract value \( \tau \), the following are equivalent:

  \[ \alpha(v) = \tau \]
Galois Connection

- Galois connection means that if we want to relate a concrete value \( v \) and abstract value \( \tau \), the following are equivalent:
  - \( \alpha(v) = \tau \)
  - \( v \in \gamma(\tau) \)
Galois Connection

- Galois connection means that if we want to relate a concrete value \( v \) and abstract value \( \tau \), the following are equivalent:
  - \( \alpha(v) = \tau \)
  - \( v \in \gamma(\tau) \)

- Think of it as a well-formed abstraction
Galois Connection

- Galois connection means that if we want to relate a concrete value $v$ and abstract value $\tau$, the following are equivalent:
  
  - $\alpha(v) = \tau$
  
  - $v \in \gamma(\tau)$

- Think of it as a well-formed abstraction

- In this class, we are only interested in Galois connections
For our type system to be sound, we require that for any program, the concrete value $v$ of this program is compatible with the type $\tau$ computed.

Formally, we state this property as follows:

If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \gamma(\tau)$.

This means that the type we give to every expression always overapproximates the type of the concrete value.

We can safely rely on the static types computed.

Slogan: "Well-typed programs cannot go wrong"
Soundness

- For our type system to be sound, we require that for any program, the concrete value $v$ of this program is compatible with the type $\tau$ computed.

- Formally, we state this property as follows:
Soundness

- For our type system to be sound, we require that for any program, the concrete value \( v \) of this program is compatible with the type \( \tau \) computed.

- Formally, we state this property as follows:

\[
\text{If } E \vdash e : v \text{ and } \Gamma \vdash e : \tau, \text{ then } v \in \gamma(\tau)
\]
Soundness

- For our type system to be **sound**, we require that for any program, the concrete value $v$ of this program is compatible with the type $\tau$ computed.

- Formally, we state this property as follows:

  If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \gamma(\tau)$

- This means that the type we give to every expression always **overapproximates** the type of the concrete value.
Soundness

- For our type system to be **sound**, we require that **for any program**, the concrete value \( v \) of this program is compatible with the type \( \tau \) computed.

- Formally, we state this property as follows:

\[
    \text{If } E \vdash e : v \text{ and } \Gamma \vdash e : \tau, \text{ then } v \in \gamma(\tau)
\]

- This means that the type we give to every expression always **overapproximates** the type of the concrete value.

- We can safely rely on the static types computed.
For our type system to be sound, we require that for any program, the concrete value $v$ of this program is compatible with the type $\tau$ computed.

Formally, we state this property as follows:

If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \gamma(\tau)$

This means that the type we give to every expression always overapproximates the type of the concrete value.

We can safely rely on the static types computed.

Slogan: “Well-typed programs cannot go wrong”
Soundness Cont.

- Clearly, not every type system is sound or useful to prevent run-time errors

Therefore, we need a way to prove that a type system we design is actually sound and useful. There are many ways of proving correspondence between abstract and concrete semantics, but the most popular strategy for types is to split the problem into two:

1. Preservation: Soundness is preserved under transition rules
2. Progress: A well-typed program never "gets stuck" when executing operational semantics (no run-time errors).

Preservation states that your type system is an overapproximation while progress states that your type system is expressive enough to prevent all run-time errors.
Soundness Cont.

- Clearly, not every type system is sound or useful to prevent run-time errors

- Therefore, we need a way to prove that a type system we design is actually sound and useful.
Soundness Cont.

- Clearly, not every type system is sound or useful to prevent run-time errors

- Therefore, we need a way to prove that a type system we design is actually sound and useful.

- There are many ways of proving correspondence between abstract and concrete semantics, but the most popular strategy for types is to split the problem into two:
Soundness Cont.

- Clearly, not every type system is sound or useful to prevent run-time errors.

- Therefore, we need a way to prove that a type system we design is actually sound and useful.

- There are many ways of proving correspondence between abstract and concrete semantics, but the most popular strategy for types is to split the problem into two:
  1. **Preservation**: Soundness is preserved under transition rules.
Soundness Cont.

- Clearly, not every type system is sound or useful to prevent run-time errors.

- Therefore, we need a way to prove that a type system we design is actually sound and useful.

- There are many ways of proving correspondence between abstract and concrete semantics, but the most popular strategy for types is to split the problem into two:
  1. **Preservation**: Soundness is preserved under transition rules.
  2. **Progress**: A well-typed program never “gets stuck” when executing operational semantics (no run-time errors).
Soundness Cont.

- Clearly, not every type system is sound or useful to prevent run-time errors.

- Therefore, we need a way to prove that a type system we design is actually sound and useful.

- There are many ways of proving correspondence between abstract and concrete semantics, but the most popular strategy for types is to split the problem into two:
  1. Preservation: Soundness is preserved under transition rules
  2. Progress: A well-typed program never “gets stuck” when executing operational semantics (no run-time errors).

- Preservation states that your type system is an overapproximation while progress states that your type system is expressive enough to prevent all run-time errors.
How to Prove Preservation

- Preservation: If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \Gamma(\tau)$ (or equivalently $\alpha(v) = \tau$)
How to Prove Preservation

- Preservation: If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \Gamma(\tau)$ (or equivalently $\alpha(v) = \tau$)

- Preservation must be argued inductively, specifically via structural induction on the program expressions
How to Prove Preservation

- Preservation: If \( E \vdash e : v \) and \( \Gamma \vdash e : \tau \), then \( v \in \Gamma(\tau) \) (or equivalently \( \alpha(v) = \tau \))

- Preservation must be argued \textit{inductively}, specifically via \textbf{structural induction} on the program expressions
  - We first need to argue preservation for all the base cases:
How to Prove Preservation

▶ Preservation: If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \Gamma(\tau)$ (or equivalently $\alpha(v) = \tau$)

▶ Preservation must be argued **inductively**, specifically via **structural induction** on the program expressions
  ▶ We first need to argue preservation for all the base cases:
    Base case: rules with no $\vdash$ in their hypotheses
How to Prove Preservation

- Preservation: If $E \vdash e : v$ and $\Gamma \vdash e : \tau$, then $v \in \Gamma(\tau)$ (or equivalently $\alpha(v) = \tau$)

- Preservation must be argued **inductively**, specifically via **structural induction** on the program expressions
  - We first need to argue preservation for all the base cases: Base case: rules with no $\vdash$ in their hypotheses
  - Then, for the inductive rules, we assume that preservation holds for all subexpressions and prove that it holds for the current expression.
How to Prove Preservation

- Preservation: If \( E \vdash e : \tau \) and \( \Gamma \vdash e : \tau \), then \( v \in \Gamma(\tau) \) (or equivalently \( \alpha(v) = \tau \))

- Preservation must be argued inductively, specifically via structural induction on the program expressions
  - We first need to argue preservation for all the base cases:
    Base case: rules with no \( \vdash \) in their hypotheses
  - Then, for the inductive rules, we assume that preservation holds for all subexpressions and prove that it holds for the current expression.

- This is a very powerful proof technique!
Let’s prove preservation of our type system, first without identifiers and let bindings:
Let’s prove *preservation* of our type system, first without identifiers and let bindings:

**Base case 1:**

\[
\begin{align*}
\text{integer } i & \quad \text{integer } i \\
E \vdash i : i & \quad \Gamma \vdash i : \text{Int}
\end{align*}
\]
Proving Preservation

Let’s prove preservation of our type system, first without identifiers and let bindings:

Base case 1:

\[
\begin{align*}
\text{integer } i & \quad \text{integer } i \\
E \vdash i : i & \quad \Gamma \vdash i : Int
\end{align*}
\]

Need to prove that \(\alpha(i) = Int\)
Let’s prove preservation of our type system, first without identifiers and let bindings:

Base case 1:

\[
\begin{align*}
\text{integer } i & \quad \text{integer } i \\
E \vdash i : i & \quad \Gamma \vdash i : \text{Int}
\end{align*}
\]

Need to prove that \( \alpha(i) = \text{Int} \)

\( \Rightarrow \) This follows directly from the hypothesis that \( i \) is an integer
Proving Preservation

- Base case 2:

\[
\begin{align*}
\text{string } s & \quad \text{string } s \\
E \vdash s : s & \quad \Gamma \vdash s : \text{String}
\end{align*}
\]

Also follows immediately that \( \alpha(s) = \text{String} \)
Proving Preservation

- Inductive case 1:

\[
\begin{align*}
E \vdash S_1 & : i_1 \\
E \vdash S_2 & : i_2 \\
\hline
E \vdash S_1 + S_2 & : i_1 + i_2
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash S_1 & : \text{Int} \\
\Gamma \vdash S_2 & : \text{Int} \\
\hline
\Gamma \vdash S_1 + S_2 & : \text{Int}
\end{align*}
\]

By the inductive hypothesis we know that \(\alpha(i_1) = \text{Int}\) and \(\alpha(i_2) = \text{Int}\). Since the value \(i_1 + i_2\) is also an integer, \(\alpha(i_1 + i_2) = \text{Int}\).
Proving Preservation

- Inductive case 1:

\[
\begin{align*}
E & \vdash S_1 : i_1 \\
E & \vdash S_2 : i_2 \\
\hline
E & \vdash S_1 + S_2 : i_1 + i_2
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash S_1 : Int \\
\Gamma & \vdash S_2 : Int \\
\hline
\Gamma & \vdash S_1 + S_2 : Int
\end{align*}
\]

- By the inductive hypothesis we know that \( \alpha(i_1) = \text{Int} \) and \( \alpha(i_2) = \text{Int} \). Since the value \( i_1 + i_2 \) is also an integer, \( \alpha(i_1 + i_2) = \text{Int} \)
Proving Preservation

- Inductive case 2:

\[
\begin{align*}
E & \vdash S_1 : s_1 \\
E & \vdash S_2 : s_2 \\
E & \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash S_1 : \text{String} \\
\Gamma & \vdash S_2 : \text{String} \\
\Gamma & \vdash S_1 :: S_2 : \text{String}
\end{align*}
\]
Proving Preservation

- Inductive case 2:

\[
\begin{align*}
E & \vdash S_1 : s_1 \\
E & \vdash S_2 : s_2 \\
\overline{E \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash S_1 : \text{String} \\
\Gamma & \vdash S_2 : \text{String} \\
\overline{\Gamma \vdash S_1 :: S_2 : \text{String}}
\end{align*}
\]

- By the **inductive hypothesis** we know that \(\alpha(s_1) = \text{String}\) and \(\alpha(s_2) = \text{String}\). Since the value \(\text{concat}(s_1, s_2)\) is also a string, \(\alpha(\text{concat}(s_1, s_2)) = \text{String}\)
Proving Preservation with Identifiers

▶ But what about the two rules involving identifiers?

\[
\begin{align*}
\text{identifier } \text{id} & : E(\text{id}) \\
\text{id} & : \Gamma(\text{id}) \\
E \vdash S_1 : e_1 & \\
E[\text{id} \leftarrow e_1] \vdash S_2 : e_2 \\
E \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : e_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash S_1 : \tau_1 & \\
\tau = \tau_1 & \\
\Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3 \\
\Gamma \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : \tau_3 \\
\end{align*}
\]

To prove the base case, we need to know that the values in \(\Gamma\) and \(E\) agree.

Definition: Concrete environment \(E\) and abstract environment \(\Gamma\) agree if for any identifier \(x\):

\[
\Gamma(\text{id}) = \alpha(E(\text{id}))
\]

Therefore, we first need to prove agreement before showing the preservation of the identifier rules.
Proving Preservation with Identifiers

But what about the two rules involving identifiers?

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash id : E(id)$</td>
<td>$\Gamma \vdash id : \Gamma(id)$</td>
</tr>
</tbody>
</table>

$E \vdash S_1 : e_1$

$E[id \leftarrow e_1] \vdash S_2 : e_2$

$E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2$

$\Gamma \vdash S_1 : \tau_1$

$\tau = \tau_1$

$\Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3$

$\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3$

To prove the base case, we need to know that the values in $\Gamma$ and $E$ agree.
Proving Preservation with Identifiers

But what about the two rules involving identifiers?

\[
\begin{align*}
\text{identifier} & \quad id \\
E \vdash id : E(id) & \quad \Gamma \vdash id : \Gamma(id) \\
\end{align*}
\]

\[
\begin{align*}
E \vdash S_1 : e_1 & \quad \Gamma \vdash S_1 : \tau_1 \\
E[id \leftarrow e_1] \vdash S_2 : e_2 & \quad \tau = \tau_1 \\
E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2 & \quad \Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \\
\end{align*}
\]

To prove the base case, we need to know that the values in \( \Gamma \) and \( E \) agree.

Definition: Concrete environment \( E \) and abstract environment \( \Gamma \) agree if for any identifier \( x \) \( \Gamma(x) = \alpha(E(x)) \), written as \( \Gamma \sim E \).
Proving Preservation with Identifiers

But what about the two rules involving identifiers?

\[
\begin{align*}
\text{identifier } id & \quad \text{identifier } id \\
E \vdash id : E(id) & \quad \Gamma \vdash id : \Gamma(id)
\end{align*}
\]

\[
\begin{align*}
E \vdash S_1 : e_1 & \quad \Gamma \vdash S_1 : \tau_1 \\
E[id \leftarrow e_1] \vdash S_2 : e_2 & \quad \tau = \tau_1 \\
E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2 & \quad \Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \\
\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3
\end{align*}
\]

To prove the base case, we need to know that the values in \(\Gamma\) and \(E\) agree.

Definition: Concrete environment \(E\) and abstract environment \(\Gamma\) agree if for any identifier \(x\) \(\Gamma(x) = \alpha(E(x))\), written as \(\Gamma \sim E\)

Therefore, we first need to prove agreement before showing the preservation of the identifier rules.
Proving Agreement

- Fortunately, proving agreement of $E$ and $\Gamma$ is easy, again by induction, on the number of mappings in $E$ and $\Gamma$
Proving Agreement

- Fortunately, proving agreement of $E$ and $\Gamma$ is easy, again by induction, on the number of mappings in $E$ and $\Gamma$

- Base case: $E$ and $\Gamma$ are empty: $\Rightarrow$ they trivially agree
Proving Agreement

- Fortunately, proving agreement of $E$ and $\Gamma$ is easy, again by induction, on the number of mappings in $E$ and $\Gamma$

- Base case: $E$ and $\Gamma$ are empty: $\Rightarrow$ they trivially agree

- Clearly, rules that do not change $E$ or $\Gamma$ cannot break agreement.
Proving Agreement

- Fortunately, proving agreement of $E$ and $\Gamma$ is easy, again by induction, on the number of mappings in $E$ and $\Gamma$.

- Base case: $E$ and $\Gamma$ are empty: $\Rightarrow$ they trivially agree.

- Clearly, rules that do not change $E$ or $\Gamma$ cannot break agreement.

- Therefore, we only have to prove agreement for the following rule:

\[
\begin{align*}
E \vdash S_1 : e_1 \\
E[\text{id} \leftarrow e_1] \vdash S_2 : e_2 \\
\hline
E \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : e_2
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash S_1 : \tau_1 \\
\tau = \tau_1 \\
\Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3 \\
\hline
\Gamma \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : \tau_3
\end{align*}
\]
Proving Agreement

\[
\begin{align*}
E & \vdash S_1 : e_1 \\
E[\text{id} \leftarrow e_1] & \vdash S_2 : e_2 \\
E & \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : e_2
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash S_1 : \tau_1 \\
\tau & = \tau_1 \\
\Gamma[\text{id} \leftarrow \tau] & \vdash S_2 : \tau_3 \\
\Gamma & \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : \tau_3
\end{align*}
\]

Here, assuming preservation, we know that \( \alpha(e_1) = \tau \). By the inductive hypothesis, we also know that \( \Gamma \sim E \).
Proving Agreement

\[
\begin{align*}
E \vdash S_1 : e_1 \\
E[id \leftarrow e_1] \vdash S_2 : e_2 \\
\Gamma \vdash S_1 : \tau_1 \\
\tau = \tau_1 \\
\Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \\
\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3
\end{align*}
\]

- Here, assuming preservation, we know that \( \alpha(e_1) = \tau \). By the inductive hypothesis, we also know that \( \Gamma \sim E \).

- Therefore, we also know that \( \Gamma[id \leftarrow \tau] \sim E[id \leftarrow e_1] \)
Here, assuming preservation, we know that $\alpha(e_1) = \tau$. By the inductive hypothesis, we also know that $\Gamma \sim E$.

Therefore, we also know that $\Gamma[id \leftarrow \tau] \sim E[id \leftarrow e_1]$

Important: We proved agreement in the inductive case assuming preservation!
Now, we can assume agreement when proving preservation:

- Base case:
  \[ \Gamma \vdash \text{id} : \Gamma(\text{id}) \]\n
This follows immediately since by our assumption \( \Gamma \sim E \).
Now, we can assume agreement when proving preservation:

Base case:

\[
\begin{align*}
\text{identifier } id & \quad \text{identifier } id \\
E \vdash id : E(id) & \quad \Gamma \vdash id : \Gamma(id)
\end{align*}
\]
Now, we can assume agreement when proving preservation:

Base case:

\[
\begin{align*}
\text{identifier } id & \quad \text{identifier } id \\
E \vdash id : E(id) & \quad \Gamma \vdash id : \Gamma(id)
\end{align*}
\]

This follows immediately since by our assumption \(\Gamma \sim E\).
Proving Preservation with Identifiers cont.

- Inductive case:

\[
\begin{align*}
E \vdash S_1 : e_1 \\
E[\text{id} \leftarrow e_1] \vdash S_2 : e_2 \\
E \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : e_2
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash S_1 : \tau_1 \\
\tau = \tau_1 \\
\Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3 \\
\Gamma \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : \tau_3
\end{align*}
\]

By the inductive hypothesis, we know that \( \alpha(e_2) = \tau_3 \). This is what we want to prove.

Observe: We combined agreement and preservation for this proof to work.

The preservation proof works assuming that agreement holds.

The agreement proof works assuming that preservation holds.

As long as both properties hold initially, this is fine!
Proving Preservation with Identifiers cont.

- Inductive case:

\[
\begin{align*}
E & \vdash S_1 : e_1 \\
E[\text{id} \leftarrow e_1] & \vdash S_2 : e_2 \\
E & \vdash \text{let id : } \tau = S_1 \text{ in } S_2 : e_2
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash S_1 : \tau_1 \\
\tau & = \tau_1 \\
\Gamma[\text{id} \leftarrow \tau] & \vdash S_2 : \tau_3 \\
\Gamma & \vdash \text{let id : } \tau = S_1 \text{ in } S_2 : \tau_3
\end{align*}
\]

- By the inductive hypothesis, we know that \(\alpha(e_2) = \tau_3\). This is what we want to prove.
Proving Preservation with Identifiers cont.

- Inductive case:

\[
\frac{E \vdash S_1 : e_1 \quad \Gamma \vdash S_1 : \tau_1 \quad \tau = \tau_1}{E \vdash let \ id : \tau = S_1 \ in \ S_2 : e_2 \quad \Gamma \vdash let \ id : \tau = S_1 \ in \ S_2 : \tau_3}
\]

\[
E[\text{id} \leftarrow e_1] \vdash S_2 : e_2 \\
\Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3
\]

- By the inductive hypothesis, we know that \( \alpha(e_2) = \tau_3 \). This is what we want to prove.

- Observe: We combined agreement and preservation for this proof to work.
Proving Preservation with Identifiers cont.

- Inductive case:

  \[
  \begin{align*}
  & E \vdash S_1 : e_1 \\
  & E[id \leftarrow e_1] \vdash S_2 : e_2 \\
  & \frac{\Gamma \vdash S_1 : \tau_1}{E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2}
  \end{align*}
  \]

  \[
  \begin{align*}
  & \tau = \tau_1 \\
  & \Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \\
  & \frac{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}{\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3}
  \end{align*}
  \]

- By the inductive hypothesis, we know that \( \alpha(e_2) = \tau_3 \). This is what we want to prove.

- Observe: We combined agreement and preservation for this proof to work.
  - The preservation proof works assuming that agreement holds.
Proving Preservation with Identifiers cont.

- Inductive case:

\begin{align*}
E \vdash S_1 : e_1 & \quad \Gamma \vdash S_1 : \tau_1 \\
E[\text{id} \leftarrow e_1] \vdash S_2 : e_2 & \quad \tau = \tau_1 \\
E \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : e_2 & \quad \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3 \\
\end{align*}

- By the inductive hypothesis, we know that \( \alpha(e_2) = \tau_3 \). This is what we want to prove.

- **Observe:** We combined agreement and preservation for this proof to work.
  - The preservation proof works **assuming** that agreement holds
  - The agreement proof works **assuming** that preservation holds
Proving Preservation with Identifiers cont.

- Inductive case:

\[\begin{align*}
E \vdash S_1 : e_1 & \quad \Gamma \vdash S_1 : \tau_1 \\
E[\text{id} \leftarrow e_1] \vdash S_2 : e_2 & \quad \tau = \tau_1 \\
E \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2 & \quad \Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3 \\
\end{align*}\]

- By the inductive hypothesis, we know that \(\alpha(e_2) = \tau_3\). This is what we want to prove.

- Observe: We combined agreement and preservation for this proof to work.
  - The preservation proof works assuming that agreement holds
  - The agreement proof works assuming that preservation holds

- As long as both properties hold initially, this is fine!
On to Progress

- We have now shown preservation of our type system
On to Progress

- We have now shown preservation of our type system

- Intuitively: We now know that the abstract value we compute will always overapproximate the concrete value for any program
On to Progress

- We have now shown preservation of our type system

- Intuitively: We now know that that abstract value we compute will always overapproximate the concrete value for any program

- Now, we want to prove that our type system is powerful enough to prevent run-time type errors
On to Progress

▶ We have now shown preservation of our type system

▶ Intuitively: We now know that that abstract value we compute will always overapproximate the concrete value for any program

▶ Now, we want to prove that our type system is powerful enough to prevent run-time type errors

▶ Or more formally, our operational semantics never “get stuck”
On to Progress

- We have now shown preservation of our type system

- Intuitively: We now know that that abstract value we compute will always overapproximate the concrete value for any program

- Now, we want to prove that our type system is powerful enough to prevent run-time type errors

- Or more formally, our operational semantics never “get stuck”

- Progress: We need to prove that every program that can be typed under our typing rules will not not “get stuck” in the operational semantics
On to Progress

- We have now shown *preservation* of our type system

- Intuitively: We now know that that abstract value we compute will always overapproximate the concrete value for any program

- Now, we want to prove that our type system is powerful enough to prevent run-time type errors

- Or more formally, our operational semantics never “get stuck”

- **Progress:** We need to prove that every program that can be typed under our typing rules will not not “get stuck” in the operational semantics

- Progress is a very strong property that few real type systems obey!
Proving Progress

- We will again prove progress by **structural induction**:
We will again prove progress by **structural induction**:

- **Base case 1: Integer**

  
  \[
  \begin{align*}
  E & \vdash i : i \\
  \Gamma & \vdash i : \text{Int}
  \end{align*}
  \]

  Clearly, if \( i \) types as an integer, the corresponding operational semantics rule applies.

- **Base case 2: String**

  \[
  \begin{align*}
  E & \vdash s : s \\
  \Gamma & \vdash s : \text{String}
  \end{align*}
  \]

  Clearly, if \( s \) types as a string, the corresponding operational semantics rule applies.
Proving Progress

- We will again prove progress by **structural induction**:
  - Base case 1: Integer

\[
\begin{align*}
\text{integer } i & \quad \text{integer } i \\
\frac{}{E \vdash i : i} & \quad \frac{}{\Gamma \vdash i : \text{Int}}
\end{align*}
\]

Clearly, if \( i \) types as an integer, the corresponding operational semantics rule applies.
We will again prove progress by \textit{structural induction}:

- **Base case 1: Integer**

  \[
  \frac{\text{integer } i}{E \vdash i : i} \quad \frac{\text{integer } i}{\Gamma \vdash i : \text{Int}}
  \]

  Clearly, if \( i \) types as an integer, the corresponding operational semantics rule applies.

- **Base case 2: String**

  \[
  \frac{\text{string } s}{E \vdash s : s} \quad \frac{\text{string } s}{\Gamma \vdash s : \text{String}}
  \]
We will again prove progress by structural induction:

- **Base case 1: Integer**

  \[
  \frac{\text{integer } i}{E \vdash i : i} \quad \frac{\text{integer } i}{\Gamma \vdash i : \text{Int}}
  \]

  Clearly, if \( i \) types as an integer, the corresponding operational semantics rule applies.

- **Base case 2: String**

  \[
  \frac{\text{string } s}{E \vdash s : s} \quad \frac{\text{string } s}{\Gamma \vdash s : \text{String}}
  \]

  Clearly, if \( s \) types as a string, the corresponding operational semantics rule applies.
Proving Progress

- Base case 3: Identifier

\[
\begin{align*}
\text{identifier } id & \quad \text{identifier } id \\
\frac{}{E \vdash id : E(id)} & \quad \frac{}{\Gamma \vdash id : \Gamma(id)}
\end{align*}
\]
Base case 3: Identifier

\[
\begin{align*}
\text{identifier } id & \quad \text{identifier } id \\
\frac{}{} & \quad \frac{}{}
\end{align*}
\]

\[\frac{E \vdash id : E(id)}{\Gamma \vdash id : \Gamma(id)}\]

Assuming agreement, we know that if the mapping \( id \mapsto \tau \) is present in \( \Gamma \), the mapping \( id \mapsto v \) is present in \( E \). Since this is the only hypothesis (precondition) of the operational semantics rule, it must therefore always apply in all well-types programs.
Proving Progress

- Inductive case 1:

\[
egin{align*}
E & \vdash S_1 : i_1 \\
E & \vdash S_2 : i_2 \\
\hline
E \vdash S_1 + S_2 : i_1 + i_2
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash S_1 : \text{Int} \\
\Gamma & \vdash S_2 : \text{Int} \\
\hline
\Gamma \vdash S_1 + S_2 : \text{Int}
\end{align*}
\]

We know from the inductive hypothesis that the evaluation of $S_1$ and $S_2$ will never get stuck. We also know from preservation that the expressions $S_1$ and $S_2$ must evaluate to integers if they are typed $\text{Int}$, therefore the operational semantics rule for plus will always apply since the hypotheses only require that $i_1$ and $i_2$ are integers.
Proving Progress

- **Inductive case 1:**

\[
\frac{E \vdash S_1 : i_1 \quad \Gamma \vdash S_1 : Int}{E \vdash S_1 + S_2 : i_1 + i_2}
\]

We know from the inductive hypothesis that the evaluation of \(S_1\) and \(S_2\) will never get stuck. We also know from preservation that the expressions \(S_1\) and \(S_2\) must evaluate to integers if they are typed \(\text{Int}\), therefore the operational semantics rule for plus will always apply since the hypotheses only require that \(i_1\) and \(i_2\) are integers.
Proving Progress

- Inductive case 2:

\[
\begin{align*}
E \vdash S_1 : s_1 \\
E \vdash S_2 : s_2 \\
\hline
E \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash S_1 : \text{String} \\
\Gamma \vdash S_2 : \text{String} \\
\hline
\Gamma \vdash S_1 :: S_2 : \text{String}
\end{align*}
\]

We know from the inductive hypothesis that the evaluation of \(S_1\) and \(S_2\) will never get stuck. We also know from preservation that the expressions \(S_1\) and \(S_2\) must evaluate to strings if they are typed \(\text{String}\), therefore the operational semantics rule for concatenation will always apply since the hypotheses only require that \(s_1\) and \(s_2\) are strings.
Proving Progress

- Inductive case 2:

\[
\begin{align*}
  & E \vdash S_1 : s_1 \\
  & E \vdash S_2 : s_2 \\
  \hline
  E \vdash S_1 :: S_2 : \text{concat}(s_1, s_2)
\end{align*}
\]

\[
\begin{align*}
  & \Gamma \vdash S_1 : \text{String} \\
  & \Gamma \vdash S_2 : \text{String} \\
  \hline
  \Gamma \vdash S_1 :: S_2 : \text{String}
\end{align*}
\]

We know from the inductive hypothesis that the evaluation of \( S_1 \) and \( S_2 \) will never get stuck. We also know from preservation that the expressions \( S_1 \) and \( S_2 \) must evaluate to strings if they are typed String, therefore the operational semantics rule for concatenation will always apply since the hypotheses only require that \( s_1 \) and \( s_2 \) are strings.
Proving Progress

- Inductive case 3:

\[
\begin{align*}
E & \vdash S_1 : e_1 \\
E \left[ \text{id} \leftarrow e_1 \right] & \vdash S_2 : e_2 \\
E & \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : e_2
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash S_1 : \tau_1 \\
\tau & = \tau_1 \\
\Gamma \left[ \text{id} \leftarrow \tau \right] & \vdash S_2 : \tau_3 \\
\Gamma & \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3
\end{align*}
\]

Here, we know from the inductive hypothesis that \( E \vdash S_1 : e_1 \) and \( E \left[ \text{id} \leftarrow e_1 \right] \vdash S_2 : e_2 \) will not get stuck since they are well-typed. Therefore, this rule will also always apply.
Inductive case 3:

\[
\begin{align*}
E & \vdash S_1 : e_1 \\
E[\text{id} \leftarrow e_1] & \vdash S_2 : e_2 \\
\frac{}{E \vdash \text{let} \ id : \tau = S_1 \ \text{in} \ S_2 : e_2}
\end{align*}
\]

Here, we know from the inductive hypothesis that \( E \vdash S_1 : e_1 \) and \( E[\text{id} \leftarrow e_1] \vdash S_2 : e_2 \) will not get stuck since they are well-typed. Therefore, this rule will also always apply.
Preservation + Progress

- We now proved both preservation and progress of our small type system on the let language.
Preservation + Progress

- We now proved both preservation and progress of our small type system on the let language.

- Important Point: You can only prove progress and preservation of a type system with respect to an operational semantics.
Preservation + Progress

- We now proved both preservation and progress of our small type system on the let language.

- **Important Point:** You can only prove progress and preservation of a type system with respect to an operational semantics.

- Poofs of preservation and progress are always by structural induction.
Preservation + Progress

- We now proved both preservation and progress of our small type system on the let language.

- **Important Point:** You can only prove progress and preservation of a type system with respect to an operational semantics

- Poofs of preservation and progress are always by structural induction

- If you have an environment, you usually need to show agreement to prove preservation
Preservation + Progress

- We now proved both preservation and progress of our small type system on the let language.

- **Important Point:** You can only prove progress and preservation of a type system *with respect to an operational semantics*.

- Poofs of preservation and progress are always by structural induction.

- If you have an environment, you usually need to show agreement to prove preservation.

- These proofs tend to always follow the same pattern, so follow this strategy on homeworks/exams.
Adding the Lambda to our language

Let us add the lambda construct to the let-language. I will call this the lambda-language:

\[
S' \rightarrow \text{integer} \mid \text{string} \mid \text{identifier} \\
\mid S_1 + S_2 \mid S_1 :: S_2 \\
\mid \text{let } id : \tau = S_1 \text{ in } S_2 \\
\mid \lambda x : \tau. S_1 \\
\mid (S_1 \ S_2)
\]

\[
\tau \rightarrow \text{Int} \mid \text{String} \mid \tau_1 \rightarrow \tau_2
\]
Adding the Lambda to our language

Let us add the lambda construct to the let-language. I will call this the lambda-language:

\[
S \rightarrow \text{integer} | \text{string} | \text{identifier} \\
| S_1 + S_2 | S_1 :: S_2 \\
| \text{let id : } \tau = S_1 \text{ in } S_2 \\
| \lambda x : \tau. S_1 \\
| (S_1 \ S_2)
\]

\[
\tau \rightarrow \text{Int} | \text{String} | \tau_1 \rightarrow \tau_2
\]

The operational semantics of the new constructs are as follows:

\[
\begin{align*}
E \vdash S_1 : \lambda x : \tau. e \\
E \vdash S_2 : e_2 \\
E \vdash e[e_2/x] : e_r
\end{align*}
\]

\[
E \vdash \lambda x : \tau. S_1 : \lambda x : \tau . S_1 \\
E \vdash (S_1 \ S_2) : e_r
\]
Typing rules for lambda and Application

- Lambda:

\[
\Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2 \\
\Gamma \vdash \lambda x : \tau_1 . S_1 : \tau_1 \rightarrow \tau_2
\]
Typing rules for lambda and Application

- Lambda:
  \[
  \frac{
    \Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2
  }{
    \Gamma \vdash \lambda x : \tau_1. S_1 : \tau_1 \rightarrow \tau_2
  }
  \]

- Application:
  \[
  \frac{
    \Gamma \vdash S_1 : \tau_1 \rightarrow \tau_2
    \quad
    \Gamma \vdash S_2 : \tau_1
  }{
    \Gamma \vdash (S_1 \ S_2) : \tau_2
  }
  \]

Observe that these almost exactly correspond to the operational semantics!

But there is one difference: The body of the let is type checked at the definition, but only evaluated at the application.
Typing rules for lambda and Application

▶ Lambda:

\[
\frac{\Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2}{\Gamma \vdash \lambda x : \tau_1.S_1 : \tau_1 \rightarrow \tau_2}
\]

▶ Application:

\[
\begin{align*}
\Gamma &\vdash S_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma &\vdash S_2 : \tau_1 \\
\Gamma &\vdash (S_1 S_2) : \tau_2
\end{align*}
\]

▶ Observe that these almost exactly correspond to the operational semantics!
Typing rules for lambda and Application

- **Lambda:**
  \[
  \frac{\Delta[x \leftarrow \tau_1] \vdash S_1 : \tau_2}{\Delta \vdash \lambda x : \tau_1 . S_1 : \tau_1 \rightarrow \tau_2}
  \]

- **Application:**
  \[
  \frac{\Delta \vdash S_1 : \tau_1 \rightarrow \tau_2 \quad \Delta \vdash S_2 : \tau_1}{\Delta \vdash (S_1 S_2) : \tau_2}
  \]

- Observe that these almost exactly correspond to the operational semantics!

- But there is one difference: The body of the let is type checked at the definition, but only evaluated at the application.
Preservation for lambda

Lambda:

\[
\frac{E \vdash \lambda x : \tau. S_1 : \lambda x : \tau . S_1}{\Gamma \vdash \lambda x : \tau_1. S_1 : \tau_1 \rightarrow \tau_2}
\]

First, we observe that if \( \Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2 \) holds, we know by our inductive hypothesis that \( \alpha(E \vdash S_1[v/x]) = \tau_2 \) for any value \( v \) of type \( \tau_1 \). Therefore, the type of this rule is \( \tau_1 \rightarrow \tau_2 \).
Preservation for lambda

- Lambda:

\[
\begin{array}{c}
E \vdash \lambda x : \tau . S_1 : \lambda x : \tau . S_1 \\
\Gamma \vdash \alpha(E \vdash S_1[v/x]) = \tau_2 \\
\end{array}
\]

- First, we observe that if \( \Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2 \) holds, we know by our inductive hypothesis that \( \alpha(E \vdash S_1[v/x]) = \tau_2 \) for any value \( v \) of type \( \tau_1 \). Therefore, the type of this rule is \( \tau_1 \rightarrow \tau_2 \).
Preservation for Application

▶ Application:

\[
\begin{align*}
E & \vdash S_1 : \lambda x : \tau. e \\
E & \vdash S_2 : e_2 \\
E & \vdash e[e_2/x] : e_r \\
E & \vdash (S_1 S_2) : e_r
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash S_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma & \vdash S_2 : \tau_1 \\
\Gamma & \vdash (S_1 S_2) : \tau_2
\end{align*}
\]
Preservation for Application

- **Application:**

  \[
  \begin{align*}
  & E \vdash S_1 : \lambda x : \tau. e \\
  & E \vdash S_2 : e_2 \\
  & E \vdash e[e_2/x] : e_r \\
  & \Gamma \vdash S_1 : \tau_1 \rightarrow \tau_2 \\
  & \Gamma \vdash S_2 : \tau_1 \\
  & \Gamma \vdash (S_1 S_2) : \tau_2 \\
  \end{align*}
  \]

- First, we observe by our inductive hypothesis that if the type of \( S_1 \) is \( \tau_1 \rightarrow \tau_2 \), the first hypothesis in the concrete rule must always apply. Second, by the inductive hypothesis we know that \( \alpha(e_2) = \tau_1 \). Since the type of \( S_1 \) is \( \tau_1 \rightarrow \tau_2 \), we can therefore safely conclude that \( \alpha(e_r) = \tau_2 \).
Question: Why could we not formulate the typing rules for lambda and application symmetric to the operational semantics?
Question: Why could we not formulate the typing rules for lambda and application symmetric to the operational semantics?

Answer: Because if we try to type check the body of a lambda at the application site, we have no way of knowing the name of the variable bound in this lambda statement.
Question: Why could we not formulate the typing rules for lambda and application symmetric to the operational semantics?

Answer: Because if we try to type check the body of a lambda at the application site, we have no way of knowing the name of the variable bound in this lambda statement.

This is typical: When typing functions, we usually always examine the function body before it is used.
Progress and Preservation in Real Languages

- **Shocking News:** Many type systems obey neither progress or preservation!
Progress and Preservation in Real Languages

- **Shocking News**: Many type systems obey neither progress or preservation!

- **Example**: C, C++
Progress and Preservation in Real Languages

- **Shocking News:** Many type systems obey neither progress or preservation!

- **Example:** C, C++

- **More Shocking News:** Very few type systems obey progress!
Progress and Preservation in Real Languages

- **Shocking News:** Many type systems obey neither progress or preservation!
  - **Example:** C, C++

- **More Shocking News:** Very few type systems obey progress!
  - **Example:** Java
Progress and Preservation in Real Languages

- **Shocking News:** Many type systems obey neither progress or preservation!

- **Example:** C, C++

- **More Shocking News:** Very few type systems obey progress!

- **Example:** Java

- But progress is a very useful property, even if it can often only be argued for some classes of run-time errors
Conclusion

- We saw how to give typing rules
Conclusion

- We saw how to give typing rules
- We proved progress and preservation of a type system
Conclusion

- We saw how to give typing rules
- We proved progress and preservation of a type system
- Next time: Polymorphism