CS345H: Programming Languages

Lecture 12: Type Inference

Thomas Dillig
Introduction

- So far when we studied typing, we always assumed that the programmer annotated some types
Introduction

- So far when we studied typing, we always assumed that the programmer annotated some types

- **Example:** We gave types to let bindings and lambda variables in class
Introduction

- So far when we studied typing, we always assumed that the programmer annotated some types

- **Example:** We gave types to let bindings and lambda variables in class

- But annotating types can be cumbersome!
Introduction

- So far when we studied typing, we always assumed that the programmer annotated some types

- **Example:** We gave types to let bindings and lambda variables in class

- But annotating types can be cumbersome!

- Anyone who has ever written C++ code can really empathize: 
  
  ```
  vector<Map<int, string>> ::const_iterator it...
  ```
Type Inference

- **Goal of type inference:** Automatically deduce the most general type for each expression
Type Inference

- **Goal of type inference:** Automatically deduce the most general type for each expression

- **Two key points:**
Type Inference

- **Goal of type inference:** Automatically deduce the most general type for each expression

- **Two key points:**
  1. Automatically inferring types: This means the programmer has to write no types, but still gets all the benefit from static typing
Goal of type inference: Automatically deduce the most general type for each expression

Two key points:

1. Automatically inferring types: This means the programmer has to write no types, but still gets all the benefit from static typing

2. Inferring the most general type: This means we want to infer polymorphic types whenever possible
Type System

Here is the type system we used in the lambda language:

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbol</th>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>$i$</td>
<td>$\Gamma \vdash i : \text{Int}$</td>
</tr>
<tr>
<td>string</td>
<td>$s$</td>
<td>$\Gamma \vdash s : \text{String}$</td>
</tr>
<tr>
<td>identifier</td>
<td>$id$</td>
<td>$\Gamma \vdash id : \Gamma(id)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1 + S_2$</td>
<td>$\Gamma \vdash S_1 + S_2 : \text{Int}$</td>
</tr>
<tr>
<td>$S_1 :: S_2$</td>
<td>$\Gamma \vdash S_1 :: S_2 : \text{String}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type Inference</th>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1 : \tau_1$</td>
<td>$\Gamma \vdash S_1 : \tau_1$</td>
</tr>
<tr>
<td>$\tau = \tau_1$</td>
<td>$\tau = \tau_1$</td>
</tr>
<tr>
<td>$\Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3$</td>
<td>$\Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3$</td>
</tr>
<tr>
<td>$\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3$</td>
<td>$\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lambda Term</th>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x : \tau_1.S_1 : \tau_1 \rightarrow \tau_2$</td>
<td>$\Gamma \vdash \lambda x : \tau_1.S_1 : \tau_1 \rightarrow \tau_2$</td>
</tr>
<tr>
<td>$(S_1 S_2) : \tau_2$</td>
<td>$\Gamma \vdash (S_1 S_2) : \tau_2$</td>
</tr>
</tbody>
</table>
Type System

Here is the type system we used in the lambda language:

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>$i$</td>
<td>$\Gamma \vdash i : \text{Int}$</td>
</tr>
<tr>
<td>String</td>
<td>$s$</td>
<td>$\Gamma \vdash s : \text{String}$</td>
</tr>
<tr>
<td>Identifier</td>
<td>$id$</td>
<td>$\Gamma \vdash id : \Gamma(id)$</td>
</tr>
<tr>
<td>Sum</td>
<td>$S_1 + S_2$</td>
<td>$\Gamma \vdash S_1 + S_2 : \text{Int}$</td>
</tr>
<tr>
<td>Concatenate</td>
<td>$S_1 :: S_2$</td>
<td>$\Gamma \vdash S_1 :: S_2 : \text{String}$</td>
</tr>
<tr>
<td>Assignment</td>
<td>$id \leftarrow \tau$</td>
<td>$\Gamma \vdash \Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3$</td>
</tr>
<tr>
<td>Let</td>
<td>$\text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3$</td>
<td>$\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3$</td>
</tr>
<tr>
<td>Lambda</td>
<td>$\lambda x : \tau_1 . S_1 : \tau_1 \rightarrow \tau_2$</td>
<td>$\Gamma \vdash \lambda x : \tau_1 . S_1 : \tau_1 \rightarrow \tau_2$</td>
</tr>
<tr>
<td>Application</td>
<td>$(S_1 S_2) : \tau_2$</td>
<td>$\Gamma \vdash (S_1 S_2) : \tau_2$</td>
</tr>
</tbody>
</table>
Type Inference Example 1

But, do we actually need these type annotations to infer the type of programs?
Type Inference Example 1

- But, do we actually need these type annotations to infer the type of programs?

- Consider the following example:
  
  ```
  let f1 = lambda x.x+2 in ..
  ```
Type Inference Example 1

- But, do we actually need these type annotations to infer the type of programs?

- Consider the following example:
  let f1 = lambda x. x+2 in ..

- Here, we know that function f1 adds two to its argument
But, do we actually need these type annotations to infer the type of programs?

Consider the following example:
let f1 = lambda x.x+2 in ..

Here, we know that function f1 adds two to its argument.

We also know that plus is only defined on integers.
But, do we actually need these type annotations to infer the type of programs?

Consider the following example:

```
let f1 = lambda x.x+2 in ..
```

Here, we know that function $f1$ adds two to its argument.

We also know that plus is only defined on integers.

Therefore, the type of $f1$ must be $Int \rightarrow Int$. 
Consider the following example:

```plaintext
let f2 = lambda x. lambda y. x + y in ..
```

Here, we know that function `f2` has two (curried) arguments, `x` and `y`. We also know that plus is only defined on integers. Therefore, the type of `f2` must be `Int → Int → Int`. 

Thomas Dillig, CS345H: Programming Languages Lecture 12: Type Inference
Consider the following example:
let f2 = lambda x. lambda y. x + y in ..

Here, we know that function f2 has two (curried) arguments, x and y
Consider the following example:

```haskell
let f2 = lambda x.lambda y.x+y in ..
```

Here, we know that function $f2$ has two (curried) arguments, $x$ and $y$

We also know that plus is only defined on integers
Consider the following example:

\[
\text{let } f_2 = \lambda x.\lambda y.x+y \text{ in ..}
\]

Here, we know that function \( f_2 \) has two (curried) arguments, \( x \) and \( y \)

We also know that plus is only defined on integers

Therefore, the type of \( f_2 \) must be \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)
Consider the following example:

\[
\text{let } f_2 = \text{lambda } x.\text{lambda } y. x + 1 \text{ in } ..
\]
Consider the following example:

\[\text{let } f2 = \lambda x. \lambda y. x + 1 \text{ in } \ldots\]

Here, we know that function \( f2 \) has two (curried) arguments, \( x \) and \( y \). Therefore, the type of \( f2 \) must be \( \forall \alpha. \text{Int} \rightarrow \alpha \rightarrow \text{Int} \).
Consider the following example:
let f2 = lambda x.lambda y.x+1 in ..

Here, we know that function f2 has two (curried) arguments, x and y

We also know that plus is only defined on integers
Consider the following example:

```plaintext
let f2 = lambda x.lambda y.x+1 in ..
```

Here, we know that function $f2$ has two (curried) arguments, $x$ and $y$

We also know that plus is only defined on integers

But $f2$ will work for any type of $y$
Consider the following example:
let \( f2 = \lambda x.\lambda y.x+1 \) in ..

Here, we know that function \( f2 \) has two (curried) arguments, \( x \) and \( y \)

We also know that plus is only defined on integers

But \( f2 \) will work for any type of \( y \)

Therefore, the type of \( f2 \) must be \( \forall \alpha.\text{Int} \to \alpha \to \text{Int} \)
Now, consider the following example:

```haskell
let f2 = lambda g.(g 0) in ..
```

Here, we know that function `f2` takes a function as argument since it is applied to 0. We also know that the function `g` is applied to an integer. Therefore, the type of `g` must be `∀α.\text{Int} → α`. This means that the type of `f2` is `∀α.((\text{Int} → α) → α)`. 
Now, consider the following example:
\[
\text{let } f2 = \lambda g. (g \ 0) \text{ in ..}
\]

Here, we know that function \( f2 \) takes a function as argument since it is applied to 0.
Now, consider the following example:

```plaintext
let f2 = lambda g.(g 0) in ..
```

Here, we know that function $f2$ takes a function as argument since it is applied to 0.

We also know that the function $g$ is applied to in integer
Type Inference Example 4

Now, consider the following example:
let f2 = lambda g.(g 0) in ..

Here, we know that function f2 takes a function as argument since it is applied to 0.

We also know that the function g is applied to in integer

Therefore, the type of g must be $\forall \alpha.\text{Int} \rightarrow \alpha$
Now, consider the following example:

```haskell
let f2 = lambda g.(g 0) in ..
```

Here, we know that function $f2$ takes a function as argument since it is applied to $0$.

We also know that the function $g$ is applied to an integer.

Therefore, the type of $g$ must be $\forall \alpha. \text{Int} \to \alpha$.

This means that the type of $f2$ is $\forall \alpha. (\text{Int} \to \alpha) \to \alpha$.
Type Inference Overview

- Goal of the rest of this lecture: Develop an algorithm that can compute the most general type for any expression without any type annotations

For this, let us look at the type derivation for the following simple function:

\[ \text{lambda } x : \text{Int}. \ x + 2 \]

Here is the type derivation tree for this expression:

\[
\begin{align*}
\text{identifier } & x \\
\Gamma(\ x) & = \text{Int} \\
\Gamma[x \leftarrow \text{Int}] & \vdash x : \text{Int} \\
\Gamma[x \leftarrow \text{Int}] & \vdash 2 : \text{Int} \\
\Gamma[x \leftarrow \text{Int}] & \vdash x + 2 : \text{Int} \\
\Gamma & \vdash \lambda x : \text{Int}. \ x + 2 : \text{Int} \rightarrow \text{Int}
\end{align*}
\]
Type Inference Overview

- Goal of the rest of this lecture: Develop an algorithm that can compute the most general type for any expression without any type annotations

- For this, let us look at the type derivation for the following simple function:
  \[ \text{lambda } \ x: \text{Int}. \ x + 2 \]
Type Inference Overview

- Goal of the rest of this lecture: Develop an algorithm that can compute the most general type for any expression without any type annotations.

- For this, let us look at the type derivation for the following simple function:
  \[ \text{lambda } x:\text{Int}. x+2 \]

- Here is the type derivation tree for this expression:

\[
\begin{align*}
\text{identifier } x \\
\Gamma(x) &= \text{Int} \\
\Gamma[x \leftarrow \text{Int}] &\vdash x : \text{Int} \\
\Gamma &\vdash \lambda x : \text{Int}. x + 2 : \text{Int} \rightarrow \text{Int}
\end{align*}
\]
Big Idea: Replace the concrete type `Int` annotated with a type variable and collect all constraints on this type variable.
Type Variables

▶ **Big Idea:** Replace the *concrete type* `Int` annotated with a type variable and collect all *constraints* on this type variable.

▶ Specifically, pretend that the type of the argument is just some type variable called `a`
Type Variables

▶ Big Idea: Replace the concrete type Int annotated with a type variable and collect all constraints on this type variable.

▶ Specifically, pretend that the type of the argument is just some type variable called \( a \)

▶ And for all rules that have preconditions on \( a \), write these preconditions as constraints
Here is the type derivation tree for this expression using type variable $a$:

```
\[
\begin{align*}
\textit{identifier } x \\
\Gamma(x) = a \\
\Gamma[x \leftarrow a] \vdash x : a \\
\end{align*}
\]

$\vdash a = \text{Int}$

```
\[
\begin{align*}
\text{integer } 2 \\
\Gamma[x \leftarrow a] \vdash 2 : \text{Int} \\
\end{align*}
\]

```
\[
\begin{align*}
\Gamma[x \leftarrow a] & \vdash x + 2 : \text{Int} \\
\end{align*}
\]

```
\[
\begin{align*}
\Gamma & \vdash \lambda x : a. x + 2 : a \rightarrow \text{Int} \\
\end{align*}
\]
Here is the type derivation tree for this expression using type variable $a$:

\[
\begin{align*}
\text{identifier } x \\
\Gamma(x) &= a \\
\Gamma[x \leftarrow a] &\vdash x : a \\
\Gamma[x \leftarrow a] &\vdash 2 : \text{Int} \\
\Gamma[x \leftarrow a] &\vdash x + 2 : \text{Int} \\
\Gamma &\vdash \lambda x : a. x + 2 : a \rightarrow \text{Int}
\end{align*}
\]

Observe that we have one additional precondition on the plus rule: The type variable $a$ must be equal to Int for this rule to apply.
Here is the type derivation tree for this expression using type variable $a$:

$$
\begin{align*}
\text{identifier } x \\
\Gamma(x) &= a \\
\Gamma[x \leftarrow a] &\vdash x : a \\
\Gamma[x \leftarrow a] &\vdash 2 : \text{Int} \\
\Gamma[x \leftarrow a] &\vdash x + 2 : \text{Int} \\
\Gamma &\vdash \lambda x:a.x + 2 : a \rightarrow \text{Int}
\end{align*}
$$

Observe that we have one additional precondition on the plus rule: The type variable $a$ must be equal to Int for this rule to apply.

We now obtain the type: $a \rightarrow \text{Int}$ and the constraint $a = \text{Int}$.
Here is the type derivation tree for this expression using type variable \( a \):

\[
\begin{array}{c}
\text{identifier } x \\
\Gamma(x) = a \\
\Gamma[x \leftarrow a] \vdash x : a \\
\Gamma[x \leftarrow a] \vdash 2 : \text{Int} \\
\Gamma \vdash \lambda x : a . x + 2 : a \rightarrow \text{Int}
\end{array}
\]

Observe that we have one additional precondition on the plus rule: The type variable \( a \) must be equal to \( \text{Int} \) for this rule to apply.

We now obtain the type: \( a \rightarrow \text{Int} \) and the constraint \( a = \text{Int} \)

Final type:
Type Variables Cont.

- Here is the type derivation tree for this expression using type variable a:

\[
\begin{align*}
\text{identifier } x \\
\Gamma(x) &= a \\
\Gamma[x \leftarrow a] &\vdash x : a \\
\end{align*}
\]

\[
\begin{align*}
a &= \text{Int} \\
\Gamma[x \leftarrow a] &\vdash 2 : \text{Int} \\
\Gamma[x \leftarrow a] &\vdash x + 2 : \text{Int} \\
\end{align*}
\]

\[
\Gamma \vdash \lambda x : a. x + 2 : a \rightarrow \text{Int}
\]

- Observe that we have one additional precondition on the plus rule: The type variable a must be equal to Int for this rule to apply.

- We now obtain the type: \( a \rightarrow \text{Int} \) and the constraint \( a = \text{Int} \)

- Final type: \( \text{Int} \rightarrow \text{Int} \)
In this example, we dealt with not knowing the type of $x$ in the following way:
Type Variables in Typing Rules

- In this example, we dealt with not knowing the type of $x$ in the following way:
  - We introduced a type variable $a$ for the type of $x$
Type Variables in Typing Rules

- In this example, we dealt with not knowing the type of \( x \) in the following way:
  - We introduced a type variable \( a \) for the type of \( x \)
  - Every time a rule uses the type of \( x \), we use \( x \)
Type Variables in Typing Rules

- In this example, we dealt with not knowing the type of $x$ in the following way:
  - We introduced a type variable $a$ for the type of $x$
  - Every time a rule uses the type of $x$, we use $x$
  - Since the plus rule has the precondition that both operands must be of type Int, we introduced a constraint $a = \text{Int}$
Type Variables in Typing Rules

- In this example, we dealt with not knowing the type of $x$ in the following way:
  - We introduced a type variable $a$ for the type of $x$
  - Every time a rule uses the type of $x$, we use $x$
  - Since the plus rule has the precondition that both operands must be of type Int, we introduced a constraint $a = \text{Int}$
  - After we typed the expression, we had a the type $a \rightarrow \text{Int}$ and the constraint $a = \text{Int}$
In this example, we dealt with not knowing the type of $x$ in the following way:

- We introduced a type variable $a$ for the type of $x$.

- Every time a rule uses the type of $x$, we use $x$.

- Since the plus rule has the precondition that both operands must be of type Int, we introduced a constraint $a = \text{Int}$.

- After we typed the expression, we had a the type $a \rightarrow \text{Int}$ and the constraint $a = \text{Int}$.

- **Solving** the type with respect to the collected constraint yields: $\text{Int} \rightarrow \text{Int}$.
Generalizing this Example

- This strategy generalizes!
Generalizing this Example

- This strategy generalizes!
- We will introduce type variables for every type annotation
Generalizing this Example

- This strategy generalizes!
- We will introduce type variables for every type annotation
- We will collect constraints on type variables during type checking
Generalizing this Example

- This strategy generalizes!
- We will introduce type variables for every type annotation
- We will collect constraints on type variables during type checking
- We will end up with a type containing type variables
Generalizing this Example

- This strategy generalizes!

- We will introduce type variables for every type annotation

- We will collect constraints on type variables during type checking

- We will end up with a type containing type variables

- We will solve this type with respect to the collected constraints
Generalizing our typing rules

- The base cases stay unchanged:

\[
\begin{align*}
\text{integer } i & \quad \rightarrow \quad \Gamma \vdash i : \text{Int} \\
\text{string } s & \quad \rightarrow \quad \Gamma \vdash s : \text{String} \\
\text{identifier } id & \quad \rightarrow \quad \Gamma \vdash id : \Gamma(id)
\end{align*}
\]

- When type checking plus, we now collect constraints on the operands:

\[
\Gamma \vdash S_1 : \tau_1 \quad \Gamma \vdash S_2 : \tau_2
\]

\[
\tau_1 = \text{Int}, \tau_2 = \text{Int}
\]

\[
\Gamma \vdash S_1 + S_2 : \text{Int}
\]

- The lines marked in red are constraints.

- Specifically, this rule now succeeds as long as \(S_1\) and \(S_2\) evaluate to any type, we simply collect constraints on the types \(\tau_1\) and \(\tau_2\) to be processed later.
Generalizing our typing rules

- The base cases stay unchanged:

\[
\begin{align*}
\Gamma \vdash i : \text{Int} \\
\Gamma \vdash s : \text{String} \\
\Gamma \vdash id : \Gamma(id)
\end{align*}
\]

- When type checking plus, we now collect constraints on the operands:

\[
\begin{align*}
\Gamma \vdash S_1 : \tau_1 \\
\Gamma \vdash S_2 : \tau_2 \\
\tau_1 &= \text{Int}, \tau_2 = \text{Int}
\end{align*}
\]

\[
\Gamma \vdash S_1 + S_2 : \text{Int}
\]
Generalizing our typing rules

▶ The base cases stay unchanged:

\[
\begin{align*}
\Gamma \vdash i : \text{Int} \\
\Gamma \vdash s : \text{String} \\
\Gamma \vdash id : \Gamma(id)
\end{align*}
\]

▶ When type checking plus, we now collect constraints on the operands:

\[
\begin{align*}
\Gamma \vdash S_1 : \tau_1 \\
\Gamma \vdash S_2 : \tau_2 \\
\tau_1 = \text{Int}, \tau_2 = \text{Int}
\end{align*}
\]

\[
\Gamma \vdash S_1 + S_2 : \text{Int}
\]

▶ The lines marked in red are constraints.
Generalizing our typing rules

- The base cases stay unchanged:

  \[
  \Gamma \vdash i : \text{Int} \quad \Gamma \vdash s : \text{String} \quad \Gamma \vdash id : \Gamma(id)
  \]

- When type checking plus, we now collect constraints on the operands:

  \[
  \Gamma \vdash S_1 : \tau_1 \\
  \Gamma \vdash S_2 : \tau_2 \\
  \tau_1 = \text{Int}, \tau_2 = \text{Int} \\
  \Gamma \vdash S_1 + S_2 : \text{Int}
  \]

- The lines marked in red are constraints.

- Specifically, this rule now succeeds as long as \( S_1 \) and \( S_2 \) evaluate to any type, we simply collect constraints on the types \( \tau_1 \) and \( \tau_2 \) to be processed later
Generalizing our typing rules

Let’s move on to the typing rule for concatenation:

\[
\begin{align*}
\Gamma &\vdash S_1 : \tau_1 \\
\Gamma &\vdash S_2 : \tau_2 \\
\tau_1 &= String, \tau_2 = String \\
\hline
\Gamma &\vdash S_1 :: S_2 : String
\end{align*}
\]
Let’s move on to the typing rule for concatenation:

\[ \Gamma \vdash S_1 : \tau_1 \]
\[ \Gamma \vdash S_2 : \tau_2 \]
\[ \tau_1 = String, \tau_2 = String \]
\[ \Gamma \vdash S_1 :: S_2 : String \]

The lines marked in red are again constraints.
Generalizing our typing rules

- Let’s move on to the typing rule for concatenation:

\[
\begin{align*}
\Gamma &\vdash S_1 : \tau_1 \\
\Gamma &\vdash S_2 : \tau_2 \\
\tau_1 &= \text{String}, \tau_2 = \text{String} \\
\hline
\Gamma &\vdash S_1 :: S_2 : \text{String}
\end{align*}
\]

- The lines marked in red are again constraints.

- Again, this rule now succeeds as long as \( S_1 \) and \( S_2 \) evaluate to any type, we simply collect constraints on the types \( \tau_1 \) and \( \tau_2 \) to be processed later.
Let’s move on to the typing rule for let:

\[
\begin{align*}
\Gamma[id \leftarrow a] & \vdash S_1 : a \quad (a \text{ fresh}) \\
\Gamma[id \leftarrow a] & \vdash S_2 : \tau \\
\hline
\Gamma & \vdash \text{let } id = S_1 \text{ in } S_2 : \tau
\end{align*}
\]

Here, all we do is introduce a fresh type variable to capture the (unknown) type of id.
Let’s move on to the typing rule for let:

\[
\begin{align*}
\Gamma[\text{id} \leftarrow a] & \vdash S_1 : a \quad (a \text{ fresh}) \\
\Gamma[\text{id} \leftarrow a] & \vdash S_2 : \tau \\
\hline
\Gamma & \vdash \text{let } \text{id} = S_1 \text{ in } S_2 : \tau
\end{align*}
\]

Here, all we do is introduce a fresh type variable to capture the (unknown) type of id.
The Let Case

Let’s move on to the typing rule for let:

\[
\begin{align*}
\Gamma[\text{id} \leftarrow a] & \vdash S_1 : a \quad (a \text{ fresh}) \\
\Gamma[\text{id} \leftarrow a] & \vdash S_2 : \tau \\
\hline
\Gamma & \vdash \text{let} \ \text{id} = S_1 \ \text{in} \ S_2 : \tau
\end{align*}
\]

Here, all we do is introduce a fresh type variable to capture the (unknown) type of id.

Observe that this case only introduces a type variable, but does not add any constraints.
Let’s move on to the typing rule for lambda:

\[ \frac{\Gamma[x \leftarrow a] \vdash S_1 : \tau \ (a \text{ fresh})}{\Gamma \vdash \lambda x. S_1 : a \rightarrow \tau} \]
The Lambda Case

Let’s move on to the typing rule for lambda:

\[ \Gamma[x \leftarrow a] \vdash S_1 : \tau \quad (a \text{ fresh}) \]
\[ \Gamma \vdash \lambda x. S_1 : a \rightarrow \tau \]

Here, again we introduce a fresh type variable to capture the (unknown) type of \( x \).
Let’s move on to the typing rule for lambda:

\[ \Gamma[x \leftarrow a] \vdash S_1 : \tau \quad (a \text{ fresh}) \]
\[ \Gamma \vdash \lambda x.S_1 : a \rightarrow \tau \]

Here, again we introduce a fresh type variable to capture the (unknown) type of x.

We also use this type variable in the return type
Now the only rule missing so far is application:

\[
\Gamma \vdash S_1 : \tau_1 \\
\Gamma \vdash S_2 : \tau_2 \\
\tau_1 = \tau_2 \rightarrow a \quad (a \text{ fresh})
\]

\[
\Gamma \vdash (S_1 S_2) : a
\]
Application

- Now the only rule missing so far is application:

  \[ \Gamma \vdash S_1 : \tau_1 \]
  \[ \Gamma \vdash S_2 : \tau_2 \]
  \[ \tau_1 = \tau_2 \rightarrow a \quad (a \text{ fresh}) \]
  \[ \frac{}{\Gamma \vdash (S_1 \; S_2) : a} \]

- Here, we again introduce a fresh type variable \( a \)
Now the only rule missing so far is application:

\[
\begin{align*}
\Gamma & \vdash S_1 : \tau_1 \\
\Gamma & \vdash S_2 : \tau_2 \\
\tau_1 = \tau_2 \rightarrow a & \quad (a \text{ fresh}) \\
\hline
\Gamma & \vdash (S_1 \ S_2) : a
\end{align*}
\]

Here, we again introduce a fresh type variable \(a\)

In this rule, this type variable encodes the return type of the application
Example 1

- Let’s use these new rules to derive the typing judgment and constraints on some examples:
  \( \lambda x. x + 2 \)
Example 1

- Let’s use these new rules to derive the typing judgment and constraints on some examples:
  \[
  \lambda x. x + 2
  \]

- Type derivation:

\[
\frac{
\begin{array}{c}
\text{identifier } x \\
\Gamma(x) = a_1 \\
\end{array}
}{
\Gamma[x \leftarrow a_1] \vdash x : a_1}
\]

\[
\frac{
\begin{array}{c}
\text{integer } 2 \\
\Gamma[x \leftarrow \text{Int}] \vdash 2 : \text{Int} \\
\end{array}
}{
\Gamma[x \leftarrow a_1] \vdash x + 2 : \text{Int}}
\]

\[
\frac{
\Gamma[x \leftarrow a_1] \vdash x + 2 : \text{Int}
}{
\Gamma \vdash \lambda x. x + 2 : a_1 \rightarrow \text{Int}}
\]

\[
a_1 = \text{Int}, \text{Int} = \text{Int}
\]
Example 1

- Let’s use these new rules to derive the typing judgment and constraints on some examples:
  \( \text{lambda } x. x + 2 \)

- Type derivation:

\[
\begin{align*}
\text{identifier } x & \quad \text{integer } 2 \\
\Gamma(x) = a_1 & \quad a_1 = \text{Int}, \text{Int} = \text{Int}
\end{align*}
\]

\[
\begin{array}{c}
\Gamma [x \leftarrow a_1] \vdash x : a_1 \\
\Gamma [x \leftarrow \text{Int}] \vdash 2 : \text{Int}
\end{array}
\]

\[
\begin{array}{c}
\Gamma [x \leftarrow a_1] \vdash x + 2 : \text{Int}
\end{array}
\]

\[
\Gamma \vdash \lambda x. x + 2 : a_1 \rightarrow \text{Int}
\]

- Final Type: \( a_1 \rightarrow \text{Int} \) under constraints \( a_1 = \text{Int}, \text{Int} = \text{Int} \)
Example 1 Cont

What does this type mean? $a_1 \rightarrow Int$ under constraints $a_1 = Int, Int = Int$
Example 1 Cont

- What does this type mean? $a_1 \to \text{Int}$ under constraints $a_1 = \text{Int}, \text{Int} = \text{Int}$

- We want to solve this type, i.e., substitute everything known from the constraints as much as possible.
What does this type mean? $a_1 \rightarrow \text{Int}$ under constraints $a_1 = \text{Int}, \text{Int} = \text{Int}$

We want to solve this type, i.e., substitute everything known from the constraints as much as possible.

Goal of Solving: Deduce final type with no constraints
What does this type mean? $a_1 \rightarrow Int$ under constraints

$a_1 = Int, Int = Int$

We want to solve this type, i.e., substitute everything known from the constraints as much as possible.

Goal of Solving: Deduce final type with no constraints

Solving this type yields $Int \rightarrow Int$
Example 2

What about the following recursive function? (This function does not terminate, but this is unimportant for this example)

```
let f = lambda x.(f x) in f
```

Final Type: $a_1$ under constraint $a_1 = a_2 \rightarrow a_3$
Example 2

- What about the following recursive function? (This function does not terminate, but this is unimportant for this example)
  \[
  \text{let } f = \lambda x.(f\ x) \text{ in } f
  \]

- Type derivation:

\[
\begin{align*}
\Gamma[f \leftarrow a_1][x \leftarrow a_2] & \vdash f : a_1 \\
\Gamma[f \leftarrow a_1][x \leftarrow a_2] & \vdash x : a_2 \\
\frac{a_1 = a_2 \rightarrow a_3}{\Gamma[f \leftarrow a_1][x \leftarrow a_2] \vdash (f\ x) : a_3} \\
\frac{\Gamma[f \leftarrow a_1] \vdash \lambda x.(f\ x) : a_1}{\Gamma[f \leftarrow a_1][x \leftarrow a_2] \vdash \lambda x.(f\ x) : a_1} \\
\frac{\Gamma \vdash \text{let } f = \lambda x.(f\ x) \text{ in } f}{\Gamma \vdash \text{let } f = \lambda x.(f\ x) \text{ in } f : a_1}
\end{align*}
\]
Example 2

- What about the following recursive function? (This function does not terminate, but this is unimportant for this example)
  \[ \text{let } f = \text{lambda } x.(f x) \text{ in } f \]

- Type derivation:

  \[
  \begin{align*}
  \Gamma[f \leftarrow a_1][x \leftarrow a_2] & \vdash f : a_1 \\
  \Gamma[f \leftarrow a_1][x \leftarrow a_2] & \vdash x : a_2 \\
  a_1 & = a_2 \rightarrow a_3 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \Gamma[f \leftarrow a_1][x \leftarrow a_2] & \vdash (f \ x) : a_3 \\
  \Gamma[f \leftarrow a_1] & \vdash \lambda x.(f \ x) : a_1 \\
  \Gamma[f \leftarrow a_1] & \vdash : a_1 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \Gamma[f \leftarrow a_1][x \leftarrow a_2] & \vdash \text{in } f : a_1 \\
  \end{align*}
  \]

- Final Type: \( a_1 \) under constraint \( a_1 = a_2 \rightarrow a_3 \)
Example 2 Cont

- Recall function: let f = lambda x.(f x) in f
Example 2 Cont

- Recall function: `let f = lambda x.(f x) in f`

- Final Type: $a_1$ under constraint $a_1 = a_2 \rightarrow a_3$, but what does this final type mean?
Example 2 Cont

- Recall function: let f = lambda x.(f x) in f

- Final Type: $a_1$ under constraint $a_1 = a_2 \rightarrow a_3$, but what does this final type mean?

- First of all, observe that we can solve this type and these constraints.
Example 2 Cont

- Recall function: `let f = lambda x.(f x) in f`

- Final Type: $a_1$ under constraint $a_1 = a_2 \rightarrow a_3$, but what does this final type mean?

- First of all, observe that we can solve this type and these constraints.

- This yields $a_2 \rightarrow a_3$
Example 2 Cont

- Recall function: \( \text{let } f = \lambda x. (f \ x) \text{ in } f \)

- Final Type: \( a_1 \) under constraint \( a_1 = a_2 \rightarrow a_3 \), but what does this final type mean?

- First of all, observe that we can solve this type and these constraints.

- This yields \( a_2 \rightarrow a_3 \)

- Here, since the solution still includes type variables, we found a polymorphic type!
Example 2 Cont

- Recall function: let \( f = \lambda x. (f \ x) \) in \( f \)

- Final Type: \( a_1 \) under constraint \( a_1 = a_2 \rightarrow a_3 \), but what does this final type mean?

- First of all, observe that we can solve this type and these constraints.

- This yields \( a_2 \rightarrow a_3 \)

- Here, since the solution still includes type variables, we found a polymorphic type!

- Here, the type is \( \forall \alpha_1. \forall \alpha_2. \alpha_1 \rightarrow \alpha_2 \)
Example 2 Cont

- Recall function: \( f = \lambda x. (f \ x) \) in \( f \)

- Final Type: \( a_1 \) under constraint \( a_1 = a_2 \rightarrow a_3 \), but what does this final type mean?

- First of all, observe that we can solve this type and these constraints.

- This yields \( a_2 \rightarrow a_3 \)

- Here, since the solution still includes type variables, we found a polymorphic type!

- Here, the type is \( \forall \alpha_1. \forall \alpha_2. \alpha_1 \rightarrow \alpha_2 \)

- We will omit the quantifier from type variables and assume that any type variable is implicitly universally quantified
Example 3

- Let’s look at the following expression
  "duck" + 7

We derived type \( \text{Int} \) under constraints \( \text{String} = \text{Int}, \text{Int} = \text{Int} \).

These constraints are unsatisfiable!

This means that the expression cannot be typed.
Example 3

- Let’s look at the following expression
  "duck" + 7

- Type derivation:

\[
\begin{align*}
\Gamma & \vdash "duck" : String \\
\Gamma & \vdash 7 : Int \\
\text{String} &= \text{Int}, \text{Int} = \text{Int} \\
\hline
\Gamma & \vdash "duck" + 7 : Int
\end{align*}
\]
Example 3

- Let’s look at the following expression
  "duck" + 7

- Type derivation:

  \[ \Gamma \vdash "duck" : String \]
  \[ \Gamma \vdash 7 : Int \]
  \[ String = Int, Int = Int \]

  \[ \Gamma \vdash "duck" + 7 : Int \]

- We derived type \( Int \) under constraints \( String = Int, Int = Int \)
Example 3

- Let’s look at the following expression
  "duck" + 7

- Type derivation:

\[
\Gamma \vdash \text{"duck" : String} \\
\Gamma \vdash 7 : \text{Int} \\
String = \text{Int}, \text{Int} = \text{Int} \\
\Gamma \vdash \text{"duck" + 7 : Int}
\]

- We derived type \text{Int} under constraints \text{String = Int, Int = Int}

- These constraints are unsatisfiable!
Example 3

- Let’s look at the following expression
  "duck" + 7

- Type derivation:

\[
\begin{align*}
\Gamma & \vdash "duck" : String \\
\Gamma & \vdash 7 : Int \\
\text{String} & = \text{Int}, \text{Int} = \text{Int} \\
\hline
\Gamma & \vdash "duck" + 7 : Int
\end{align*}
\]

- We derived type Int under constraints String = Int, Int = Int

- These constraints are unsatisfiable!

- This means that the expression cannot be typed
Observe that we have split the problem of type inference into two separate problems:
Type Inference Structure

- Observe that we have split the problem of type inference into two separate problems:
  1. **Constraint Inference**: In this step, we apply the typing rules to find the type (potentially in terms of type variables) and type constraints.
Type Inference Structure

- Observe that we have split the problem of type inference into two separate problems:
  1. **Constraint Inference**: In this step, we apply the typing rules to find the type (potentially in terms of type variables) and type constraints
  2. **Constraint Solving**: In this step, we solve the constraints. Either we find a (potentially polymorphic) final type or the constraints are unsatisfiable, in which case the program cannot be typed
Type Inference Structure

- Observe that we have split the problem of type inference into two separate problems:
  1. **Constraint Inference:** In this step, we apply the typing rules to find the type (potentially in terms of type variables) and type constraints
  2. **Constraint Solving:** In this step, we solve the constraints. Either we find a (potentially polymorphic) final type or the constraints are unsatisfiable, in which case the program cannot be typed

- Observe that step 1 can never get stuck! We now reject all programs that cannot be types in step 2.
Constraint Solving

- So far, we have only informally sketched what we mean by solving type constraints
Constraint Solving

- So far, we have only informally sketched what we mean by solving type constraints

- **Convention:** I will write constraints as a list with the type of the program at the bottom
Constraint Solving

- So far, we have only informally sketched what we mean by solving type constraints

- **Convention:** I will write constraints as a list with the type of the program at the bottom

- **Example:** Consider again the expression let f = lambda x.(f x) in f
Constraint Solving

- So far, we have only informally sketched what we mean by solving type constraints.

- **Convention:** I will write constraints as a list with the type of the program at the bottom.

- **Example:** Consider again the expression `let f = lambda x.(f x) in f`

- Here, the type of `f` written as list of constraints is:

  \[
  a_1 = a_2 \rightarrow a_3 \\
  a_1
  \]
Constraint Solving

- **Definition:** A solution to a system of type constraints is a substitution $\sigma$ mapping type variables to types such that all type constraints are satisfied.

- We discovered one solution, $\alpha_1 \rightarrow \alpha_2$ for the system $a_1 = a_2 \rightarrow a_3$.

- Substitution: $\sigma = \{ a_1 \leftarrow \alpha_1, a_2 \leftarrow \alpha_2, a_3 \leftarrow (\alpha_1 \rightarrow \alpha_2) \}$.

- But the following is also a solution: Int $\rightarrow$ Int.

- Substitution: $\sigma = \{ a_1 \leftarrow \text{Int}, a_2 \leftarrow \text{Int}, a_3 \leftarrow (\text{Int} \rightarrow \text{Int}) \}$. 
**Constraint Solving**

- **Definition:** A solution to a system of type constraints is a substitution $\sigma$ mapping type variables to types such that all type constraints are satisfied.

- We discovered one solution, $\alpha_1 \rightarrow \alpha_2$ for the system:

  \[
  a_1 = a_2 \rightarrow a_3 \\
  a_1
  \]
Constraint Solving

Definition: A solution to a system of type constraints is a substitution $\sigma$ mapping type variables to types such that all type constraints are satisfied.

We discovered one solution, $\alpha_1 \rightarrow \alpha_2$ for the system

$$a_1 = a_2 \rightarrow a_3$$

$\alpha_1$

Substitution: $\sigma = \{a_1 \leftarrow \alpha_1, a_2 \leftarrow \alpha_2, a_3 \leftarrow (\alpha_1 \rightarrow \alpha_2)\}$
Definition: A solution to a system of type constraints is a substitution $\sigma$ mapping type variables to types such that all type constraints are satisfied.

We discovered one solution, $\alpha_1 \rightarrow \alpha_2$ for the system:

$$a_1 = a_2 \rightarrow a_3$$
$$a_1$$

Substitution: $\sigma = \{a_1 \leftarrow \alpha_1, a_2 \leftarrow \alpha_2, a_3 \leftarrow (\alpha_1 \rightarrow \alpha_2)\}$

But the following is also a solution: $Int \rightarrow Int$
Constraint Solving

- **Definition:** A solution to a system of type constraints is a substitution $\sigma$ mapping type variables to types such that all type constraints are satisfied.

- We discovered one solution, $\alpha_1 \to \alpha_2$ for the system:
  
  \[
  a_1 = a_2 \to a_3 \\
  a_1
  \]

- **Substitution:** $\sigma = \{ a_1 \leftarrow \alpha_1, a_2 \leftarrow \alpha_2, a_3 \leftarrow (\alpha_1 \to \alpha_2) \}$

- But the following is also a solution: $\text{Int} \to \text{Int}$

- **Substitution:** $\sigma = \{ a_1 \leftarrow \text{Int}, a_2 \leftarrow \text{Int}, a_3 \leftarrow (\text{Int} \to \text{Int}) \}$
Constraint Solving

- And $\alpha \rightarrow \alpha$ is also a solution for

$$a_1 = a_2 \rightarrow a_3$$

$$a_1$$
Constraint Solving

- And $\alpha \rightarrow \alpha$ is also a solution for

\[
\begin{align*}
  a_1 &= a_2 \rightarrow a_3 \\
  a_1 &
\end{align*}
\]

- Substitution: $\sigma = \{ a_1 \leftarrow \alpha, a_2 \leftarrow \alpha, a_3 \leftarrow (\alpha \rightarrow \alpha) \}$

But clearly some solutions are more general than others. We want to find the most general solution, also known as the most general unifier. This can be done using unification.
Constraint Solving

▶ And $\alpha \rightarrow \alpha$ is also a solution for

$$
\begin{align*}
a_1 &= a_2 \rightarrow a_3 \\
a_1
\end{align*}
$$

▶ Substitution: $\sigma = \{ a_1 \leftarrow \alpha, a_2 \leftarrow \alpha, a_3 \leftarrow (\alpha \rightarrow \alpha) \}$

▶ But clearly some solutions are more general than others.
Constraint Solving

- And $\alpha \rightarrow \alpha$ is also a solution for

  \[
  a_1 = a_2 \rightarrow a_3 \\
  a_1
  \]

- Substitution: $\sigma = \{ a_1 \leftarrow \alpha, a_2 \leftarrow \alpha, a_3 \leftarrow (\alpha \rightarrow \alpha) \}$

- But clearly some solutions are more general than others.

- We want to find the most general solution, also known as the most general unifier.
Constraint Solving

- And $\alpha \rightarrow \alpha$ is also a solution for

  \[
  a_1 = a_2 \rightarrow a_3
  \]

  \[
  a_1
  \]

- Substitution: $\sigma = \{ a_1 \leftarrow \alpha, a_2 \leftarrow \alpha, a_3 \leftarrow (\alpha \rightarrow \alpha) \}$

- But clearly some solutions are more general than others.

- We want to find the most general solution, also known as the most general unifier.

- This can be done using unification
Constraint Solving Cont.

- First Idea: We choose a variable on left-hand side and replace all occurrences of this variable with its right-hand side. In other words, we add the substitution $x \leftarrow y$ for the equality $x = y$
Constraint Solving Cont.

- First Idea: We choose a variable on left-hand side and replace all occurrences of this variable with its right-hand side. In other words, we add the substitution $x \leftarrow y$ for the equality $x = y$

- Consider again the constraint system:

\[
a_1 = a_2 \rightarrow a_3
\]
\[
a_1
\]
First Idea: We choose a variable on left-hand side and replace all occurrences of this variable with its right-hand side. In other words, we add the substitution $x \leftarrow y$ for the equality $x = y$.

Consider again the constraint system:

$$a_1 = a_2 \rightarrow a_3$$

Here, we pick $a_1$. It’s right-hand side is $a_2 \rightarrow a_3$. If we replace all occurrences of $a_1$, we get:

$$a_2 \rightarrow a_3 = a_2 \rightarrow a_3$$

and the substitution $\sigma = \{a_1 \leftarrow (a_2 \rightarrow a_3), a_2 \leftarrow a_2, a_3 \leftarrow a_3\}$
Then, drop all trivial constraints:

\[ a_2 \rightarrow a_3 \]

with substitution \( \sigma = \{ a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)
Constraint Solving Cont.

- Then, drop all trivial constraints:

  \[ a_2 \rightarrow a_3 \]

  with substitution \( \sigma = \{ a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)

- Repeat until we find a contradiction (\( \text{Int} = \text{String} \)) or there are no equalities left.
Constraint Solving Cont.

- Then, drop all trivial constraints:

\[ a_2 \rightarrow a_3 \]

with substitution \( \sigma = \{ a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)

- Repeat until we find a contradiction (\( \text{Int} = \text{String} \)) or there are no equalities left.

- In this case, we have found the most general solution.
Constraint Solving Example

Another example:

\[ a_1 = a_2 \rightarrow Int \]
\[ a_1 = String \rightarrow a_3 \]
Constraint Solving Example

- Another example:

\[
\begin{align*}
  a_1 &= a_2 \rightarrow \text{Int} \\
  a_1 &= \text{String} \rightarrow a_3
\end{align*}
\]

- Let’s pick \( a_1 \):

\[
\begin{align*}
  a_2 \rightarrow \text{Int} &= a_2 \rightarrow \text{Int} \\
  a_2 \rightarrow \text{Int} &= \text{String} \rightarrow a_3
\end{align*}
\]

with \( \sigma = \{ a_1 \leftarrow a_2 \rightarrow \text{Int}, a_2 \leftarrow a_2, a_3 \leftarrow a_3 \} \)
Constraint Solving Example

- Another example:

\[
\begin{align*}
a_1 &= a_2 \rightarrow \text{Int} \\
a_1 &= \text{String} \rightarrow a_3
\end{align*}
\]

- Let’s pick \(a_1\):

\[
\begin{align*}
a_2 \rightarrow \text{Int} &= a_2 \rightarrow \text{Int} \\
a_2 \rightarrow \text{Int} &= \text{String} \rightarrow a_3
\end{align*}
\]

with \(\sigma = \{ a_1 \leftarrow a_2 \rightarrow \text{Int}, a_2 \leftarrow a_2, a_3 \leftarrow a_3 \}\)

- Remove redundant constraints:

\[
\begin{align*}
a_2 \rightarrow \text{Int} &= \text{String} \rightarrow a_3
\end{align*}
\]

with \(\sigma = \{ a_2 \leftarrow a_2, a_3 \leftarrow a_3 \}\)
Constraint Solving Example

▶ Another example:

\[
\begin{align*}
a_1 &= \text{Int} \\
a_1 &= \text{String} \rightarrow a_3
\end{align*}
\]

▶ Let's pick \(a_1\):

\[
\begin{align*}
a_2 \rightarrow \text{Int} &= a_2 \rightarrow \text{Int} \\
a_2 \rightarrow \text{Int} &= \text{String} \rightarrow a_3
\end{align*}
\]

with \(\sigma = \{a_1 \leftarrow a_2 \rightarrow \text{Int}, a_2 \leftarrow a_2, a_3 \leftarrow a_3\}\)

▶ Remove redundant constraints:

\[
\begin{align*}
a_2 \rightarrow \text{Int} &= \text{String} \rightarrow a_3
\end{align*}
\]

with \(\sigma = \{a_2 \leftarrow a_2, a_3 \leftarrow a_3\}\)

▶ But now we are stuck, even though the final substitution is

\[
\sigma = \{a_2 \leftarrow \text{String}, a_3 \leftarrow \text{Int}, \ldots\}\]
Constraint Solving Example

Solution: Add one more rule:

Rule: If $X \rightarrow Y = W \rightarrow Z$, then add substitution $X = W$ and $Y = Z$.

Back to the example:

$a_2 \rightarrow \text{Int} = \text{String} \rightarrow a_3$

with $\sigma = \{a_2 \leftarrow a_2, a_3 \leftarrow a_3\}$

Add $s_2 \leftarrow \text{Int}$ and $a_3 \leftarrow \text{Int}$

New constraint system: $\text{String} \rightarrow \text{Int} = \text{String} \rightarrow \text{Int}$ with $\sigma = \{a_2 \leftarrow \text{String}, a_3 \leftarrow \text{Int}\}$
Constraint Solving Example

- Solution: Add one more rule:

  - Rule: If $X \rightarrow Y = W \rightarrow Z$, then add substitution $X = W$ and $Y = Z$
Constraint Solving Example

- Solution: Add one more rule:

- Rule: If $X \to Y = W \to Z$, then add substitution $X = W$ and $Y = Z$

- Back to the example:

$$a_2 \to \text{Int} = \text{String} \to a_3$$

with $\sigma = \{a_2 \leftarrow a_2, a_3 \leftarrow a_3\}$
Constraint Solving Example

Solution: Add one more rule:

Rule: If $X \rightarrow Y = W \rightarrow Z$, then add substitution $X = W$ and $Y = Z$

Back to the example:

$$a_2 \rightarrow Int = String \rightarrow a_3$$

with $\sigma = \{a_2 \leftarrow a, a_3 \leftarrow a_3\}$

Add $s_2 \leftarrow Int$ and $a_3 \leftarrow String$
Constraint Solving Example

- Solution: Add one more rule:

- Rule: If $X \rightarrow Y = W \rightarrow Z$, then add substitution $X = W$ and $Y = Z$

- Back to the example:

$$a_2 \rightarrow \text{Int} = \text{String} \rightarrow a_3$$

with $\sigma = \{ a_2 \leftarrow a_2, a_3 \leftarrow a_3 \}$

- Add $s_2 \leftarrow \text{Int}$ and $a_3 \leftarrow \text{String}$

- New constraint system:

$$\text{String} \rightarrow \text{Int} = \text{String} \rightarrow \text{Int}$$

with $\sigma = \{ a_2 \leftarrow \text{String}, a_3 \leftarrow \text{Int} \}$
Simple Unification Algorithm

- From constraints, pick one equality $a_x = e$ and apply substitution $a_x \leftarrow e$
Simple Unification Algorithm

- From constraints, pick one equality $a_x = e$ and apply substitution $a_x \leftarrow e$

- If such an equality does not exist, pick an equality of the form $X \rightarrow Y = W \rightarrow Z$ and apply substitutions $X \leftarrow W$, $Y \leftarrow Z$
Simple Unification Algorithm

- From constraints, pick one equality \( a_x = e \) and apply substitution \( a_x \leftarrow e \)

- If such an equality does not exist, pick an equality of the form \( X \rightarrow Y = W \rightarrow Z \) and apply substitutions \( X \leftarrow W, Y \leftarrow Z \)

- Repeat until we either derive a contradiction or there are not equalities left. This is a most general unifier.
Conclusion

- We have seen how we can use our typing rules to generate type constraints.
Conclusion

- We have seen how we can use our typing rules to generate type constraints.
- We looked at a simple algorithm to solve these constraints.
Conclusion

- We have seen how we can use our typing rules to generate type constraints.
- We looked at a simple algorithm to solve these constraints.
- But this algorithm is not very efficient.
Conclusion

- We have seen how we can use our typing rules to generate type constraints.
- We looked at a simple algorithm to solve these constraints.
- But this algorithm is not very efficient.
- Next time: How to perform unification efficiently and type inference in L.