Recall for last time: We are *inferring types*
Introduction

- Recall for last time: We are inferring types

- Big idea: Replace all concrete type assumptions with type variables
Introduction

- Recall for last time: We are inferring types
- Big idea: Replace all concrete type assumptions with type variables
- Collect constraints on these type variables
Recall for last time: We are inferring types

Big idea: Replace all concrete type assumptions with type variables

Collect constraints on these type variables

Find most general solution for these constraints to deduce types
Quick Refresher

- Lets quickly look again at one example:
  
  ```
  let f = lambda x.(f x) in f
  ```

  Type derivation:

  
  
  ```
  \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
  \Gamma[f \leftarrow a_1][x \leftarrow a_2] \vdash (f x) : a_3
  \Gamma[f \leftarrow a_1][x \leftarrow a_2] \vdash \lambda x.(f x) : a_1
  \Gamma[f \leftarrow a_1][x \leftarrow a_2] f \vdash : a_1
  \Gamma \vdash let f = \lambda x.(f x) in f : a_1
  ```

  Final Type:

  ```
  a_1 under constraint a_1 = a_2 \rightarrow a_3
  ```

  This yielded constraint system

  ```
  a_1 = a_2 \rightarrow a_3
  a_1
  ```
Quick Refresher

- Lets quickly look again at one example:
  \[\text{let } f = \lambda x. (f \ x) \text{ in } f\]

- Type derivation:

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

- This yielded constraint system

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

- Let's quickly look again at one example:
  \[
  \text{let } f = \lambda x. (f \ x) \text{ in } f
  \]

- Type derivation:
  \[
  \begin{align*}
  \Gamma[f ← a_1][x ← a_2] & \vdash f : a_1 \\
  \Gamma[f ← a_1][x ← a_2] & \vdash x : a_2 \\
  a_1 = a_2 \rightarrow a_3 \\
  \overline{\Gamma[f ← a_1][x ← a_2] \vdash (f \ x) : a_3} & \quad \Gamma[f ← a_1]f \vdash : a_1 \\
  \Gamma[f ← a_1] & \vdash \lambda x. (f \ x) : a_1 \\
  \overline{\Gamma[f ← a_1] \vdash \text{let } f = \lambda x. (f \ x) \text{ in } f : a_1}
  \end{align*}
  \]

- Final Type: \( a_1 \) under constraint \( a_1 = a_2 \rightarrow a_3 \)
Quick Refresher

- Lets quickly look again at one example:
  
  \[
  \text{let } f = \lambda x. (f \ x) \text{ in } f
  \]

- Type derivation:

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

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

- This yielded constraint system

  \[
  a_1 = a_2 \rightarrow a_3 \\
  a_1
  \]
Last time, we discussed two substitution rules that allow us to solve such constraints and find the most general solution.
Solving Constraints

- Last time, we discussed two substitution rules that allow us to solve such constraints and find the most general solution.
- However, the cost of this is quadratic in the number of constraints.
Solving Constraints

- Last time, we discussed two substitution rules that allow us to solve such constraints and find the most general solution.

- However, the cost of this is quadratic in the number of constraints.

- For a large program, this is prohibitive.
Solving Constraints

- Last time, we discussed two substitution rules that allow us to solve such constraints and find the most general solution.

- However, the cost of this is quadratic in the number of constraints.

- For a large program, this is prohibitive.

- **Today:** How to efficiently solve type constraint systems.
Representing Types

- Our type constraint systems are made up of the following three primitives:
Representing Types

- Our type constraint systems are made up of the following three primitives:
  1. Type constants:
Our type constraint systems are made up of the following three primitives:

1. Type constants: Int, String
Representing Types

- Our type constraint systems are made up of the following three primitives:
  1. Type constants: Int, String
  2. Type variables:

- Observe that $X \rightarrow Y$ is just in-fix notation for function $(X, Y)$.
- To solve type constraints more efficiently, we will write $X \rightarrow Y$ also as function $(X, Y)$, but this is just notation.
Representing Types

- Our type constraint systems are made up of the following three primitives:
  1. Type constants: Int, String
  2. Type variables: $\alpha_1, \alpha_2$
Representing Types

- Our type constraint systems are made up of the following three primitives:
  1. Type constants: Int, String
  2. Type variables: $\alpha_1, \alpha_2$
  3. Function Types:
Representing Types

- Our type constraint systems are made up of the following three primitives:
  1. Type constants: Int, String
  2. Type variables: $\alpha_1, \alpha_2$
  3. Function Types: $X \rightarrow Y$

Observe that $X \rightarrow Y$ is just in-fix notation for function $(X, Y)$.

To solve type constraints more efficiently, we will write $X \rightarrow Y$ also as function $(X, Y)$, but this is just notation.
Representing Types

- Our type constraint systems are made up of the following three primitives:
  1. Type constants: Int, String
  2. Type variables: $\alpha_1, \alpha_2$
  3. Function Types: $X \rightarrow Y$

- Observe that $X \rightarrow Y$ is just in-fix notation for $function(X, Y)$
Representing Types

- Our type constraint systems are made up of the following three primitives:
  1. Type constants: Int, String
  2. Type variables: $\alpha_1, \alpha_2$
  3. Function Types: $X \rightarrow Y$

- Observe that $X \rightarrow Y$ is just in-fix notation for $\text{function}(X, Y)$

- To solve type constraints more efficiently, we will write $X \rightarrow Y$ also as $\text{function}(X, Y)$, but this is just notation.
More Efficient Type Inference

- **Big Idea**: Maintain equivalence classes of types directly
More Efficient Type Inference

- **Big Idea**: Maintain equivalence classes of types directly

- **Equivalence Class**: Set of types that must be equal

Specifically, if we process constraint of the form $X = Y$, we know that $X$ and $Y$ are equal. In this case, we want to union the equivalence classes of $X$ and $Y$. Also, if $X$ and $Y$ are function types of the form $X_1 \rightarrow X_2$ and $Y_1 \rightarrow Y_2$, we also want to union $X_1$ and $Y_1$ as well as $X_2$ and $Y_2$. 
More Efficient Type Inference

- **Big Idea**: Maintain equivalence classes of types directly

- **Equivalence Class**: Set of types that must be equal

- Specifically, if we process constraint of the form $X = Y$, we know that $X$ and $Y$ are equal
More Efficient Type Inference

▶ **Big Idea:** Maintain equivalence classes of types directly

▶ **Equivalence Class:** Set of types that must be equal

▶ Specifically, if we process constraint of the form $X = Y$, we know that $X$ and $Y$ are equal

▶ In this case, we want to **union** the equivalence classes of $X$ and $Y$
More Efficient Type Inference

- **Big Idea:** Maintain equivalence classes of types directly

- **Equivalence Class:** Set of types that must be equal

- Specifically, if we process constraint of the form $X = Y$, we know that $X$ and $Y$ are equal

- In this case, we want to **union** the equivalence classes of $X$ and $Y$

- Also, if $X$ and $Y$ are function types of the form $X_1 \rightarrow X_2$ and $Y_1 \rightarrow Y_2$, we also want to union $X_1$ and $Y_1$ as well as $X_2$ and $Y_2$
Union-Find

- To maintain equivalence classes directly, we will use the union-find algorithm
Union-Find

- To maintain equivalence classes directly, we will use the union-find algorithm

- Each set of types is called an equivalence class
Union-Find

- To maintain equivalence classes directly, we will use the **union-find** algorithm

- Each set of types is called an **equivalence class**

- Each set has one element as its **representative**
Union-Find

- To maintain equivalence classes directly, we will use the *union-find* algorithm

- Each set of types is called an *equivalence class*

- Each set has one element as its *representative*

- **For type inference:** If an equivalence contains a type constant or a function type, we will always use this type as the representative.
Union-Find Cont.

- In Union-Find, we have only two operations on equivalence classes:

  1. Union $(s, t)$: This unions the equivalence classes of $s$ and $t$ into one equivalence class
  2. Find $(s)$: This returns the representative of the equivalence class of which $s$ is part of

Example: Assume following two equivalence classes (representatives in red):

- $\{\text{int}, \alpha\}$
- $\{\beta \rightarrow \gamma, \text{int}\}$

Example: Union $(\text{int}, \beta \rightarrow \gamma)$ results in new equivalence class $\{\text{int}, \alpha, \beta \rightarrow \gamma\}$

Example: Find $(\alpha) = \text{int}$
In Union-Find, we have only two operations on equivalence classes:

1. \textit{Union}(s, t): This unions the equivalence classes of $s$ and $t$ into one equivalence class

Example: Assume following two equivalence classes (representatives in red):

\{int, \alpha\}, \{\beta \rightarrow \gamma, int\}

Example: \textit{Union}(int, \beta \rightarrow \gamma) results in new equivalence class \{int, \alpha, \beta \rightarrow \gamma\}
Union-Find Cont.

- In Union-Find, we have only two operations on equivalence classes:
  1. $\text{Union}(s, t)$: This unions the equivalence classes of $s$ and $t$ into one equivalence class
  2. $\text{Find}(s)$: This returns the representative of the equivalence class of which $s$ is part of

Example: Assume following two equivalence classes (representatives in red):

\{int, α\}, \{β → γ, int\}

Example: $\text{Union}(\text{int}, β → γ)$ results in new equivalence class \{int, α, β → γ\}

Example: $\text{Find}(α) = \text{int}$
In Union-Find, we have only two operations on equivalence classes:

1. $\text{Union}(s, t)$: This unions the equivalence classes of $s$ and $t$ into one equivalence class

2. $\text{Find}(s)$: This returns the representative of the equivalence class of which $s$ is part of

Example: Assume following two equivalence classes (representatives in red): $\{\text{int}, \alpha\}, \{\beta \rightarrow \gamma, \text{int}\}$
In Union-Find, we have only two operations on equivalence classes:

1. \( \text{Union}(s, t) \): This unions the equivalence classes of \( s \) and \( t \) into one equivalence class

2. \( \text{Find}(s) \): This returns the representative of the equivalence class of which \( s \) is part of

Example: Assume following two equivalence classes (representatives in red): \( \{ \text{int}, \alpha \}, \{ \beta \rightarrow \gamma, \text{int} \} \)

Example: \( \text{Union}(\text{int}, \beta \rightarrow \gamma) \)
Union-Find Cont.

- In Union-Find, we have only two operations on equivalence classes:
  1. $\text{Union}(s, t)$: This unions the equivalence classes of $s$ and $t$ into one equivalence class
  2. $\text{Find}(s)$: This returns the representative of the equivalence class of which $s$ is part of

- Example: Assume following two equivalence classes (representatives in red): $\{\text{int}, \alpha\}, \{\beta \rightarrow \gamma, \text{int}\}$

- Example: $\text{Union}(\text{int}, \beta \rightarrow \gamma)$ results in new equivalence class $\{\text{int}, \alpha, \beta \rightarrow \gamma\}$
In Union-Find, we have only two operations on equivalence classes:

1. \( \text{Union}(s, t) \): This unions the equivalence classes of \( s \) and \( t \) into one equivalence class.

2. \( \text{Find}(s) \): This returns the representative of the equivalence class of which \( s \) is part of.

Example: Assume following two equivalence classes (representatives in red): \( \{ \text{int}, \alpha \}, \{ \beta \to \gamma, \text{int} \} \)

Example: \( \text{Union} (\text{int}, \beta \to \gamma) \) results in new equivalence class \( \{ \text{int}, \alpha, \beta \to \gamma \} \)

Example: \( \text{Find} (\alpha) = \)
In Union-Find, we have only two operations on equivalence classes:

1. \textit{Union}(s, t): This unions the equivalence classes of \(s\) and \(t\) into one equivalence class

2. \textit{Find}(s): This returns the representative of the equivalence class of which \(s\) is part of

Example: Assume following two equivalence classes (representatives in red): \(\{\text{int, } \alpha\}, \{\beta \rightarrow \gamma, \text{int}\}\)

Example: \textit{Union}(\text{int, } \beta \rightarrow \gamma) \) results in new equivalence class \(\{\text{int, } \alpha, \beta \rightarrow \gamma\}\)

Example: \textit{Find}(\alpha) = \text{int}
Union-Find Representation

- We will represent equivalence classes as DAGs.
Union-Find Representation

- We will represent equivalence classes as DAGs.

- Example: \( \{ \beta \to \gamma, \alpha \} \)
Union-Find Representation

- We will represent equivalence classes as DAGs.

- Example: \( \{ \beta \to \gamma, \alpha \} \)
Union-Find Representation

- We will represent equivalence classes as DAGs.
- Example: \( \{ \beta \rightarrow \gamma, \alpha \} \)

Conceptually, union will join the dotted areas of two equivalence classes.
Union-Find Representation

- We will represent equivalence classes as DAGs.

- Example: \( \{ \beta \rightarrow \gamma, \alpha \} \)

- Conceptually, union will join the dotted areas of two equivalence classes

- And find will return the (red) representative in this class
Union-Find Representation Cont.

- Consider the following EQs:
Consider the following EQs:
Union-Find Representation Cont.

- Consider the following EQs:

- And now consider $\text{union}(\beta \rightarrow \gamma, \text{int})$
Union-Find Representation Cont.

- Consider the following EQs:

- And now consider $\text{union}(\beta \rightarrow \gamma, \text{int})$
Question: Is this a possible solution for the type constraints?
Question: Is this a possible solution for the type constraints?

No! If a function type and a constant type ever end up in the same equivalence class, we know that the constraint system has no solution.
Question: Is this a possible solution for the type constraints?

No! If a function type and a constant type ever end up in the same equivalence class, we know that the constraint system has no solution.

We also know constraint system has no solution if \textit{Int} and \textit{String} end up in the same EQ.
Finding a Solution from the Union-Find DAG

- Assuming we end up with an consistent Union-find DAG, we can read the most general solution right of!

  - For each type variable $v$, simply return $\text{find}(v)$

  - In other words, the representative of each equivalence class is the most general solution

  - Question: Why do we always pick function types or type constants as representatives?

  - Question: What happens if a function type and a type constant are in the same equivalence class?
Finding a Solution from the Union-Find DAG

- Assuming we end up with a consistent Union-find DAG, we can read the most general solution right off!

- For each type variable $v$, simply return $\text{find}(v)$
Finding a Solution from the Union-Find DAG

- Assuming we end up with a consistent Union-find DAG, we can read the most general solution right of!

- For each type variable $v$, simply return $\text{find}(v)$

- In other words, the representative of each equivalence class is the most general solution
Finding a Solution from the Union-Find DAG

- Assuming we end up with a consistent Union-find DAG, we can read the most general solution right of!

- For each type variable $v$, simply return $find(v)$

- In other words, the representative of each equivalence class is the most general solution

- **Question**: Why do we always pick function types or type constants as representatives?
Finding a Solution from the Union-Find DAG

- Assuming we end up with a consistent Union-find DAG, we can read the most general solution right of!

- For each type variable $v$, simply return $\text{find}(v)$

- In other words, the representative of each equivalence class is the most general solution

- **Question:** Why do we always pick function types or type constants as representatives?

- **Question:** What happens if a function type and a type constant are in the same equivalence class?
Finding a Solution from the Union-Find DAG

- Example:

```
\( \alpha \) → \( \beta \) → \( \gamma \)
```

What about \( \beta \)?

Every item is in its own EQ, therefore

\( \text{find}(\beta) = \beta \)
Finding a Solution from the Union-Find DAG

Example:

How do we find solution for $\alpha$?

Every item is in its own EQ, therefore find($\beta$) = $\beta$ → $\gamma$.
Finding a Solution from the Union-Find DAG

- Example:

  ![Diagram]

  - How do we find solution for $\alpha$?
  - $\text{find}(\alpha) =$
Finding a Solution from the Union-Find DAG

- Example:

  - How do we find solution for $\alpha$?
  - $\text{find}(\alpha) = \beta \rightarrow \gamma$
Finding a Solution from the Union-Find DAG

Example:

How do we find solution for $\alpha$?

$\text{find}(\alpha) = \beta \rightarrow \gamma$

What about $\beta$?
Finding a Solution from the Union-Find DAG

Example:

- How do we find solution for $\alpha$?
- $\text{find}(\alpha) = \beta \rightarrow \gamma$
- What about $\beta$?
- Every item is in its own EQ, therefore $\text{find}(\beta) = \beta$
Using Union-Find for solving Type Inference Constraints

- Initially, all type variables, functions and type constants are in their own equivalence class

```cpp
bool unify(m, n) {
    s = find(m); t = find(n);
    if(s == t) return true;
    if(s == s1 -> s2 && t == t1 -> t2) {
        union(s, t);
        return unify(s1, t1) && unify(s2, t2);
    }
    if(is_variable(s) || is_variable(t)) {
        union(s, t); return true;
    }
    return false; //No solution to type constraints
}
```
Using Union-Find for solving Type Inference Constraints

- Initially, all type variables, functions and type constants are in their own equivalence class

- We then apply the following function to each equality in our type constraint:

```cpp
bool unify(m, n) {
    s = find(m); t = find(n);
    if(s == t) return true;
    if(s == s1 \rightarrow s2 && t == t1 \rightarrow t2) {
        union(s, t);
        return unify(s1, t1) && unify(s2, t2);
    }
    if(is_variable(s) || is_variable(t)) {
        union(s, t); return true;
    }
    return false; // No solution to type constraints
}
```
Example

Consider the following system of type constraints:

\[
\begin{align*}
\alpha \rightarrow \text{Int} & = \beta \\
\gamma \rightarrow \text{Int} & = \beta \\
\gamma & = \text{String}
\end{align*}
\]
Example

Consider the following system of type constraints:

\[ \alpha \to \text{Int} = \beta \]
\[ \gamma \to \text{Int} = \beta \]
\[ \gamma = \text{String} \]
Example

Consider the following system of type constraints:

\[
\begin{align*}
\alpha &\rightarrow \text{Int} = \beta \\
\gamma &\rightarrow \text{Int} = \beta \\
\gamma & = \text{String}
\end{align*}
\]
Example

Consider the following system of type constraints:

\[ \alpha \rightarrow \text{Int} = \beta \]
\[ \gamma \rightarrow \text{Int} = \beta \]
\[ \gamma = \text{String} \]
Consider the following system of type constraints:

\[
\begin{align*}
\alpha \rightarrow \text{Int} &= \beta \\
\gamma \rightarrow \text{Int} &= \beta \\
\gamma &= \text{String}
\end{align*}
\]
Example

Consider the following system of type constraints:

\[
\begin{align*}
\alpha \to Int &= \beta \\
\gamma \to Int &= \beta \\
\gamma &= String
\end{align*}
\]
Example

Consider the following system of type constraints:

\[
\begin{align*}
\alpha \rightarrow \text{Int} &= \beta \\
\gamma \rightarrow \text{Int} &= \beta \\
\gamma &= \text{String}
\end{align*}
\]
Example

Consider the following system of type constraints:

\[
\alpha \rightarrow \text{Int} = \beta \\
\gamma \rightarrow \text{Int} = \beta \\
\gamma = \text{String}
\]
Example Cont

\[ \alpha \rightarrow \text{Int} = \beta \]
\[ \gamma \rightarrow \text{Int} = \beta \]
\[ \gamma = \text{String} \]

Solution for \( \alpha \):

Solution for \( \beta \):

Solution for \( \gamma \):
Example Cont

\[ \alpha \to Int = \beta \]
\[ \gamma \to Int = \beta \]
\[ \gamma = String \]

- Solution for \( \alpha \): \( \text{find}(\alpha) = \)
Example Cont

\[
\alpha \rightarrow \text{Int} = \beta \\
\gamma \rightarrow \text{Int} = \beta \\
\gamma = \text{String}
\]

▶ Solution for \(\alpha\): \(\text{find}(\alpha) = \text{String}\)
Example Cont

\[ \alpha \rightarrow \text{Int} = \beta \]
\[ \gamma \rightarrow \text{Int} = \beta \]
\[ \gamma = \text{String} \]

▶ Solution for \( \alpha \): \( \text{find}(\alpha) = \text{String} \)

▶ Solution for \( \beta \):
Example Cont

\[ \alpha \rightarrow \text{Int} \quad = \quad \beta \]
\[ \gamma \rightarrow \text{Int} \quad = \quad \beta \]
\[ \gamma \quad = \quad \text{String} \]

- Solution for \( \alpha \): \( \text{find}(\alpha) = \text{String} \)
- Solution for \( \beta \): \( \text{find}(\beta) = \)
Example Cont

\[
\begin{align*}
\alpha \rightarrow \text{Int} &= \beta \\
\gamma \rightarrow \text{Int} &= \beta \\
\gamma &= \text{String}
\end{align*}
\]

▶ Solution for \(\alpha\): \(\text{find}(\alpha) = \text{String}\)

▶ Solution for \(\beta\): \(\text{find}(\beta) = \text{String} \rightarrow \text{Int}\)
Example Cont

\[ \alpha \rightarrow \text{Int} = \beta \]
\[ \gamma \rightarrow \text{Int} = \beta \]
\[ \gamma = \text{String} \]

▶ Solution for \( \alpha \): \( \text{find}(\alpha) = \text{String} \)

▶ Solution for \( \beta \): \( \text{find}(\beta) = \text{String} \rightarrow \text{Int} \)

▶ Solution for \( \gamma \):
Example Cont

\[ \alpha \rightarrow \text{Int} = \beta \]
\[ \gamma \rightarrow \text{Int} = \beta \]
\[ \gamma = \text{String} \]

- Solution for \( \alpha \): \( \text{find}(\alpha) = \text{String} \)
- Solution for \( \beta \): \( \text{find}(\beta) = \text{String} \rightarrow \text{Int} \)
- Solution for \( \gamma \): \( \text{find}(\gamma) = \)
Example Cont

\[ \alpha \rightarrow \text{Int} = \beta \]
\[ \gamma \rightarrow \text{Int} = \beta \]
\[ \gamma = \text{String} \]

\[ \text{Solution for } \alpha: \text{find}(\alpha) = \text{String} \]

\[ \text{Solution for } \beta: \text{find}(\beta) = \text{String} \rightarrow \text{Int} \]

\[ \text{Solution for } \gamma: \text{find}(\gamma) = \text{String} \]
Example 2

- Consider the following system of type constraints:

\[
\begin{align*}
\alpha &= \text{Int} \rightarrow \text{Int} \\
\alpha &= \text{String}
\end{align*}
\]
Example 2

- Consider the following system of type constraints:

\[
\alpha = \text{Int} \rightarrow \text{Int} \\
\alpha = \text{String}
\]

Conflict: Unify returns false when trying to unify \(\text{Int} \rightarrow \text{Int}\) and \(\text{String}\)

Conclusion: This system of type constraints is unsatisfiable.
Example 2

- Consider the following system of type constraints:

\[ \alpha = \text{Int} \rightarrow \text{Int} \]
\[ \alpha = \text{String} \]

Conflict: Unify returns false when trying to unify \(\text{Int} \rightarrow \text{Int}\) and \(\text{String}\)

Conclusion: This system of type constraints is unsatisfiable
Example 2

Consider the following system of type constraints:

\[ \alpha = \text{Int} \rightarrow \text{Int} \]
\[ \alpha = \text{String} \]
Example 2

Consider the following system of type constraints:

\[
\alpha = \text{Int} \rightarrow \text{Int} \\
\alpha = \text{String}
\]

Conflict: Unify returns false when trying to unify \(\text{Int} \rightarrow \text{Int}\) and \(\text{String}\).

Conclusion: This system of type constraints is unsatisfiable.
Example 2

Consider the following system of type constraints:

\[ \alpha = \text{Int} \rightarrow \text{Int} \]
\[ \alpha = \text{String} \]

Conflict: Unify returns false when trying to unify \( \text{Int} \rightarrow \text{Int} \) and \( \text{String} \).

Conclusion: This system of type constraints is unsatisfiable.
Example 2

Consider the following system of type constraints:

\[ \alpha = \text{Int} \rightarrow \text{Int} \]
\[ \alpha = \text{String} \]

Conflict: Unify returns false when trying to unify \( \text{Int} \rightarrow \text{Int} \) and \( \text{String} \)
Example 2

Consider the following system of type constraints:

\[
\alpha = \text{Int} \rightarrow \text{Int} \\
\alpha = \text{String}
\]

- **Conflict**: Unify returns false when trying to unify $\text{Int} \rightarrow \text{Int}$ and $\text{String}$

- **Conclusion**: This system of type constraints is unsatisfiable
Union-Find

- With this new approach, we can now only process each equality once.
Union-Find

- With this new approach, we can now only process each equality once.

- However, for this to be efficient, union/find must be efficient.
Union-Find

- With this new approach, we can now only process each equality once.

- However, for this to be efficient, union/find must be efficient.

- **Key result from algorithms:** It is possible to build a data structure for union-find that can find a solution to our sets of type constraints in approximately linear time.
Union-Find

- With this new approach, we can now only process each equality once.

- However, for this to be efficient, union/find must be efficient.

- **Key result from algorithms:** It is possible to build a data structure for union-find that can find a solution to our sets of type constraints in *approximately linear time*.

- You can learn about this data structure in Advance Algorithms or Isil’s class on automated logical reasoning.
With this new approach, we can now only process each equality once.

However, for this to be efficient, union/find must be efficient.

Key result from algorithms: It is possible to build a data structure for union-find that can find a solution to our sets of type constraints in approximately linear time.

You can learn about this data structure in Advance Algorithms or Isil’s class on automated logical reasoning.

But for our purposes, we will just use this data structure.
Type Inference

- If we use Union-Find, we can make type inference practical on real programs
Type Inference

- If we use Union-Find, we can make type inference practical on real programs

- This style of polymorphic type inference we studied is known as Hindley-Milner type inference
Type Inference

- If we use Union-Find, we can make type inference practical on real programs

- This style of polymorphic type inference we studied is known as Hindley-Milner type inference

- Type inference is at the core of languages such as OCAML and Haskell
Type Inference

- If we use Union-Find, we can make type inference practical on real programs

- This style of polymorphic type inference we studied is known as Hindley-Milner type inference

- Type inference is at the core of languages such as OCAML and Haskell

- Type inference is increasingly moving to main-stream languages
Type Inference

- If we use Union-Find, we can make type inference practical on real programs

- This style of polymorphic type inference we studied is known as Hindley-Milner type inference

- Type inference is at the core of languages such as OCAML and Haskell

- Type inference is increasingly moving to main-stream languages
  - New C++11 standard
Type Inference

- If we use Union-Find, we can make type inference practical on real programs
- This style of polymorphic type inference we studied is known as Hindley-Milner type inference
- Type inference is at the core of languages such as OCAML and Haskell
- Type inference is increasingly moving to main-stream languages
  - New C++11 standard
  - Java 7
Type Inference and Errors

- We saw that we can detect all errors easily when doing type inference.
We saw that we can detect all errors easily when doing type inference.

Specifically, every error resulted from unifying two equivalence classes that could not be unified.
Type Inference and Errors

- We saw that we can detect all errors easily when doing type inference.

- Specifically, every error resulted from unifying two equivalence classes that could not be unified.

- **Example:** Trying to unify \( String \) and \( \alpha \rightarrow Int \)
We saw that we can detect all errors easily when doing type inference.

Specifically, every error resulted from unifying two equivalence classes that could not be unified.

Example: Trying to unify $String$ and $\alpha \rightarrow Int$.

But how do we report this error to programmers?
Consider again the example: $String$ and $\alpha \rightarrow Int$. 
Error Reporting

- Consider again the example: $String$ and $\alpha \rightarrow Int$.

- Option 1: Output message: $String$ and $\alpha \rightarrow Int$ cannot be unified.
Error Reporting

- Consider again the example: \( \text{String and } \alpha \rightarrow \text{Int} \).

- Option 1: Output message: \( \text{String and } \alpha \rightarrow \text{Int} \) cannot be unified.

- Is this helpful?
Error Reporting

▶ Consider again the example: \( String \) and \( \alpha \rightarrow Int \).

▶ Option 1: Output message: \( String \) and \( \alpha \rightarrow Int \) cannot be unified.

▶ Is this helpful?

▶ Obvious problems:
Error Reporting

- Consider again the example: \( \text{String and } \alpha \rightarrow \text{Int} \).

- Option 1: Output message: \( \text{String and } \alpha \rightarrow \text{Int} \) cannot be unified.

- Is this helpful?

- Obvious problems:
  - Not associated with any source location
Error Reporting

- Consider again the example: $String$ and $\alpha \rightarrow Int$.

- Option 1: Output message: $String$ and $\alpha \rightarrow Int$ cannot be unified.

- Is this helpful?

- Obvious problems:
  - Not associated with any source location
  - Understanding typing errors requires understanding type inference
Error Reporting Cont.

- Improvement used in practice: Associate expression/source location with type constraint.

   Message can now at least contain the program expressions that evaluate to `String` and `α -> Int`.

   But the actual error in your program may be arbitrarily far from these locations!

   Typical OCaml error: "At line 37: Expected expression of type 'a -> 'a but found expression of type 'a -> 'b."

   To fix this, you need to understand all the reasoning steps that happened during type inference.

   Most likely, the problem did not originate at line 37!
Error Reporting Cont.

- Improvement used in practice: Associate expression/source location with type constraint.

- Message can now at least contain the program expressions that evaluate to \( \text{String} \) and \( \alpha \to \text{Int} \).
Error Reporting Cont.

- Improvement used in practice: Associate expression/source location with type constraint.

- Message can now at least contain the program expressions that evaluate to \( String \) and \( \alpha \rightarrow Int \)

- But the actual error in your program may be arbitrarily far from these locations!
Error Reporting Cont.

- Improvement used in practice: Associate expression/source location with type constraint.

- Message can now at least contain the program expressions that evaluate to `String` and `α → Int`.

- But the actual error in your program may be arbitrarily far from these locations!

- Typical OCaml error:
  
  “At line 37: Expected expression of type ‘a -> ‘a but found expression of type ‘a -> ‘b”
Error Reporting Cont.

- Improvement used in practice: Associate expression/source location with type constraint.

- Message can now at least contain the program expressions that evaluate to \( String \) and \( \alpha \rightarrow \text{Int} \).

- But the actual error in your program may be arbitrarily far from these locations!

- Typical OCaml error:
  
  “At line 37: Expected expression of type ‘a \(\rightarrow\) ‘a but found expression of type ‘a \(\rightarrow\) ‘b”

- To fix this, you need to understand all the reasoning steps that happened during type inference.
Error Reporting Cont.

- Improvement used in practice: Associate expression/source location with type constraint.

- Message can now at least contain the program expressions that evaluate to String and $\alpha \rightarrow \text{Int}$

- But the actual error in your program may be arbitrarily far from these locations!

- Typical OCaml error:
  “At line 37: Expected expression of type ‘a -> ‘a but found expression of type ‘a -> ‘b”

- To fix this, you need to understand all the reasoning steps that happened during type inference

- Most likely, the problem did not originate at line 37!
Type Annotations

- Most common technique for mitigating these difficulties: Allow type annotations
Type Annotations

- Most common technique for mitigating these difficulties: Allow type annotations

- Type annotations allow you explicitly declare types even though the compiler can infer them automatically
Type Annotations

- Most common technique for mitigating these difficulties: Allow type annotations

- Type annotations allow you explicitly declare types even though the compiler can infer them automatically

- Idea: If you encounter a type error you do not understand, you give the type you expect to the expressions involved in this error and re-run the type checker
Type Annotations

- Most common technique for mitigating these difficulties: Allow type annotations

- Type annotations allow you explicitly declare types even though the compiler can infer them automatically

- **Idea:** If you encounter a type error you do not understand, you give the type you expect to the expressions involved in this error and re-run the type checker

- You will now get a new type error in a different location
Type Annotations

- Most common technique for mitigating these difficulties: Allow type annotations

- Type annotations allow you explicitly declare types even though the compiler can infer them automatically

- **Idea:** If you encounter a type error you do not understand, you give the type you expect to the expressions involved in this error and re-run the type checker

- You will now get a new type error in a different location

- You repeat this process until you fixed your type error
Type Annotations Drawbacks

- However, this approach still has substantial drawbacks:
Type Annotations Drawbacks

- However, this approach still has substantial drawbacks:
  - You often need many annotations to find the source of type errors
Type Annotations Drawbacks

- However, this approach still has substantial drawbacks:
  - You often need many annotations to find the source of type errors
  - You can only annotate successfully if you understand polymorphic type inference
Type Annotations Drawbacks

- However, this approach still has substantial drawbacks:
  - You often need many annotations to find the source of type errors
  - You can only annotate successfully if you understand polymorphic type inference
  - You often end up with a program that is almost completely type annotated!
Type Inference in the Real World

- Despite these difficulties, there are many real languages that support full type inference.
Type Inference in the Real World

- Despite these difficulties, there are many real languages that support full type inference.

- Examples: OCaml, Haskell, F#
Type Inference in the Real World

- Despite these difficulties, there are many real languages that support full type inference.

- **Examples**: OCaml, Haskell, F#

- **Slogan on Type Inference**: The ease of dynamic typing with the speed and guarantees of a static type system
Type Inference in the Real World

- Despite these difficulties, there are many real languages that support full type inference.

- **Examples:** OCaml, Haskell, F#

- **Slogan on Type Inference:** The ease of dynamic typing with the speed and guarantees of a static type system

- This claim is true, but real problems with explaining typing errors to programmers
Type Inference in the Real World

- Despite these difficulties, there are many real languages that support full type inference.

- **Examples:** OCaml, Haskell, F#

- **Slogan on Type Inference:** The ease of dynamic typing with the speed and guarantees of a static type system

- This claim is true, but real problems with explaining typing errors to programmers

- Explaining typing errors better is also an active research area!
Type Inference in the Real World Cont.

- Alternative approach taken by more main-stream languages recently: **local type inference**
Type Inference in the Real World Cont.

- Alternative approach taken by more main-stream languages recently: local type inference

- In local type inference, types are only inferred within one function, but must be fully annotated at function boundaries.
Alternative approach taken by more main-stream languages recently: **local type inference**

In local type inference, types are only inferred within one function, but must be fully annotated at function boundaries.

**Goal:** Make it easier for programmers to diagnose type errors (and make type inference tractable in the imperative setting)
Example of local type inference

- C++ supports some forms of local type inference.
Example of local type inference

- C++ supports some forms of local type inference.
- First Example: templates

**Example:**
```cpp
pair<int, string> p(3, "duck");
```

However, if you call a function that takes a pair, the compiler will infer the template type for you in some cases:
```cpp
edit_pair(p)
```
Example of local type inference

- C++ supports some forms of local type inference.

- First Example: templates

- A STL pair is templatized over the type of the first and second element
Example of local type inference

- C++ supports some forms of local type inference.

- First Example: templates

- A STL pair is templatized over the type of the first and second element

- You declare a pair as: pair<int, string> p(3, "duck");
Example of local type inference

- C++ supports some forms of local type inference.

- First Example: templates

- A STL pair is templatized over the type of the first and second element

- You declare a pair as: `pair<int, string> p(3, "duck");`

- However, if you call a function that takes a pair, the compiler will infer the template type for you in some cases:
Example of local type inference

- C++ supports some forms of local type inference.

- First Example: templates

- A STL pair is templatized over the type of the first and second element

- You declare a pair as: `pair<int, string> p(3, "duck");`

- However, if you call a function that takes a pair, the compiler will infer the template type for you in some cases:

- Example: `edit_pair(p)` instead of `edit_pair<pair<int, string>>(p)`
Example of local type inference

- The new C++11 standard supports much more expressive local type inference
Example of local type inference

- The new C++11 standard supports much more expressive local type inference
- This is done using the auto keyword
Example of local type inference

- The new C++11 standard supports much more expressive local type inference

- This is done using the auto keyword

- **Example using iterator:**
  ```cpp
type<int> v;
...
for(vector<int>::iterator it = v.begin(); it != v.end(); it++) ...  ```
The new C++11 standard supports much more expressive local type inference.

This is done using the `auto` keyword.

**Example using iterator:**
```cpp
vector<int> v;
...
for(vector<int>::iterator it = v.begin(); it != v.end(); it++) ...
```

**Example using iterator with new auto keyword:**
```cpp
vector<int> v;
...
for(auto it = v.begin(); it != v.end(); it++) ...
```
Type Inference in C++

- The `auto` keyword really just says “do type inference on this expression and figure the type out”
Type Inference in C++

- The `auto` keyword really just says “do type inference on this expression and figure the type out”

- Very convenient, local feature that is also creeping into languages such as C# and Java
Type Inference in C++

- The `auto` keyword really just says “do type inference on this expression and figure the type out”

- Very convenient, local feature that is also creeping into languages such as C# and Java

- You will see more of this in the future
Conclusion

- We saw how to use Union-Find to make type inference scalable
Conclusion

- We saw how to use Union-Find to make type inference scalable

- This formulation is one of the classic and elegant results in programming languages, known as Hindley-Milner type inference
Conclusion

- We saw how to use Union-Find to make type inference scalable.

- This formulation is one of the classic and elegant results in programming languages, known as Hindley-Milner type inference.

- Type inference is most likely coming to your favorite language in the near future, if it is not already there!