CS312: Programming Languages

Lecture 12: Polymorphism and Type Inference

Thomas Dillig
Outline

- Finish discussion of polymorphism
- Type inference
Review

- Recall: Polymorphism allows writing generic functions can take any type as argument

Example: $\lambda \alpha. \lambda x : \alpha. x$

What is the type of this function?

$\forall \alpha. \alpha \rightarrow \alpha$

What does $(\lambda \alpha. \lambda x : \alpha. x) \text{Int}$ evaluate to?
Recall: Polymorphism allows writing generic functions can take any type as argument.

In addition, we get all the benefits of a static type system, i.e., ruling out run-time type errors.
Review

- **Recall**: Polymorphism allows writing generic functions can take any type as argument

- In addition, we get all the benefits of a static type system, i.e., ruling out run-time type errors

- **Big Idea**: Introduce type variables that range over any type
Review

- **Recall:** Polymorphism allows writing generic functions can take *any type* as argument

- In addition, we get all the benefits of a static type system, i.e., ruling out run-time type errors

- **Big Idea:** Introduce *type variables* that range over any type

- **Example:** $\Lambda \alpha. \lambda x : \alpha. x$
Review

- **Recall:** Polymorphism allows writing generic functions can take *any type* as argument

- In addition, we get all the benefits of a static type system, i.e., ruling out run-time type errors

- **Big Idea:** Introduce *type variables* that range over any type

- **Example:** $\Lambda \alpha. \lambda x : \alpha. x$

- What is the type of this function?
Review

- **Recall**: Polymorphism allows writing generic functions can take any type as argument

- In addition, we get all the benefits of a static type system, i.e., ruling out run-time type errors

- **Big Idea**: Introduce type variables that range over any type

- **Example**: $\Lambda \alpha. \lambda x : \alpha. x$

- What is the type of this function? $\forall \alpha. \alpha \rightarrow \alpha$
Review

- **Recall:** Polymorphism allows writing generic functions can take *any type* as argument
- In addition, we get all the benefits of a static type system, i.e., ruling out run-time type errors
- **Big Idea:** Introduce *type variables* that range over any type
- **Example:** \( \Lambda \alpha. \lambda x : \alpha. x \)
- What is the type of this function? \( \forall \alpha. \alpha \rightarrow \alpha \)
- What does \((\Lambda \alpha. \lambda x : \alpha. x) \text{ Int}\) evaluate to?
Polymorphic Lambda Language

\[
\begin{align*}
S & \rightarrow \text{integer} \mid \text{string} \mid \text{identifier} \\
& \quad \mid S_1 + S_2 \mid S_1 :: S_2 \\
& \quad \mid \text{let } \text{id} : \tau = S_1 \text{ in } S_2 \\
& \quad \mid \lambda x : \tau . S_1 \\
& \quad \mid \Lambda \alpha . S_1 \\
& \quad \mid (S_1 S_2) \mid (S_1 \tau) \\
\tau & \rightarrow \text{Int} \mid \text{String} \mid \tau_1 \rightarrow \tau_2 \mid \alpha \mid \forall \alpha . \tau
\end{align*}
\]
Typing rules are judgments of the form:

\[ \Delta, \Gamma \vdash e : \tau \]
Typing rules are judgments of the form:

$$\Delta, \Gamma \vdash e : \tau$$

- As before, $\Gamma$ maps identifiers to types?
Typing rules are judgments of the form:

\[ \Delta, \Gamma \vdash e : \tau \]

As before, \( \Gamma \) maps identifiers to types?

What is \( \Delta \)?
### Type Checking Polymorphic Lambda Language

- Typing rules are judgments of the form:
  \[ \Delta, \Gamma \vdash e : \tau \]

- As before, \( \Gamma \) maps identifiers to types?

- What is \( \Delta \)? well-formedness environment, maps types to \( \star \)
Typing rules are judgments of the form:

\[ \Delta, \Gamma \vdash e : \tau \]

As before, \( \Gamma \) maps identifiers to types?

What is \( \Delta \)? well-formedness environment, maps types to \( \star \)

What does it mean for type to be well-formed?
Type Checking Polymorphic Lambda Language

- Typing rules are judgments of the form:

  \[ \Delta, \Gamma \vdash e : \tau \]

- As before, \( \Gamma \) maps identifiers to types?

- What is \( \Delta \)? well-formedness environment, maps types to \(*\)

- What does it mean for type to be well-formed?

- Why do we need \( \Delta \)?
Well-formedness Judgments

- We need rules to define the well-formedness judgment
  \[ \Delta \vdash \tau : \star \]
Well-formedness Judgments

- We need rules to define the well-formedness judgment \( \Delta \vdash \tau : \star \)

- This judgment is defined inductively:
Well-formedness Judgments

- We need rules to define the well-formedness judgment $\Delta \vdash \tau : \star$

- This judgment is defined inductively:
  - Base case 1:
    $$
    \Delta \vdash \text{Int} : \star \quad \Delta \vdash \text{String} : \star
    $$
Well-formedness Judgments

- We need rules to define the well-formedness judgment
  \[ \Delta \vdash \tau : \star \]

- This judgment is defined inductively:
  - Base case 1:
    \[ \Delta \vdash \text{Int} : \star \quad \Delta \vdash \text{String} : \star \]
  - Base case 2:
    \[ \Delta \vdash \alpha : \Delta(\alpha) \]
On to the inductive rules:
On to the inductive rules:

- Inductive Case 1:
  \[ \Delta \vdash \tau_1 : \star \quad \Delta \vdash \tau_2 : \star \]
  \[ \Delta \vdash \tau_1 \to \tau_2 : \star \]
On to the inductive rules:

- **Inductive Case 1:**
  
  $$
  \frac{
  \Delta \vdash \tau_1 : \star \quad \Delta \vdash \tau_2 : \star
  }{
  \Delta \vdash \tau_1 \rightarrow \tau_2 : \star
  }
  $$

- **Inductive Case 2:**
  
  $$
  \frac{
  \Delta[\alpha \leftarrow \star] \vdash \tau : \star
  }{
  \Delta \vdash \forall \alpha.\tau : \star
  }
  $$
Well-formedness Rules Cont.

- On to the inductive rules:
  - Inductive Case 1:
    \[
    \Delta \vdash \tau_1 : \star \quad \Delta \vdash \tau_2 : \star \\
    \quad \quad \quad \Delta \vdash \tau_1 \rightarrow \tau_2 : \star
    \]
  - Inductive Case 2:
    \[
    \Delta[\alpha \leftarrow \star] \vdash \tau : \star \\
    \quad \quad \quad \Delta \vdash \forall \alpha. \tau : \star
    \]

- All this says is that if \( \Delta \vdash \tau : \star \) holds, type \( \tau \) has no free variables.
Typing Rules

▶ Let’s look at the typing rules affected by type variables:

- Function definition:
  \[ \Delta \vdash \tau_1 : \star \quad \Delta, \Gamma[x \leftarrow \tau_1] \vdash e : \tau_2 \]
  \[ \Delta, \Gamma \vdash \lambda x : \tau_1. e : \tau_1 \to \tau_2 \]

- Let binding:
  \[ \Delta \vdash \tau : \star \quad \Delta, \Gamma \vdash S_1 : \tau \quad \Delta, \Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \]
  \[ \Delta, \Gamma \vdash \text{let } id = S_1 \text{ in } S_2 : \tau_3 \]

Observe that there are two different kinds of judgments here!
Typing Rules

Let’s look at the typing rules affected by type variables:

- Function definition:

\[
\Delta \vdash \tau_1 : \star \\
\Delta, \Gamma[x \leftarrow \tau_1] \vdash e : \tau_2 \\
\Delta, \Gamma \vdash \lambda x:\tau_1.e : \tau_1 \rightarrow \tau_2
\]
Typing Rules

Let’s look at the typing rules affected by type variables:

- **Function definition:**

\[
\begin{align*}
\Delta & \vdash \tau_1 : \star \\
\Delta, \Gamma[x \leftarrow \tau_1] & \vdash e : \tau_2 \\
\Delta, \Gamma & \vdash \lambda x: \tau_1 . e : \tau_1 \rightarrow \tau_2
\end{align*}
\]
Typing Rules

Let’s look at the typing rules affected by type variables:

Function definition:

\[
\Delta \vdash \tau_1 : \star \\
\Delta, \Gamma[x \leftarrow \tau_1] \vdash e : \tau_2 \\
\Delta, \Gamma \vdash \lambda x:\tau_1 . e : \tau_1 \to \tau_2
\]

Let binding:

\[
\Delta \vdash \tau : \star \\
\Delta, \Gamma \vdash S_1 : \tau \\
\Delta, \Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \\
\Delta, \Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3
\]
Let’s look at the typing rules affected by type variables:

- **Function definition:**

  \[ \Delta \vdash \tau_1 : * \]
  \[ \Delta, \Gamma[x \leftarrow \tau_1] \vdash e : \tau_2 \]
  \[ \Delta, \Gamma \vdash \lambda x: \tau_1 . e : \tau_1 \rightarrow \tau_2 \]

- **Let binding:**

  \[ \Delta \vdash \tau : * \]
  \[ \Delta, \Gamma \vdash S_1 : \tau \]
  \[ \Delta, \Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \]
  \[ \Delta, \Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3 \]

- Observe that there are two different kinds of judgments here!
And now the typing rules for type abstractions and applications:
Typing Rules cont.

- And now the typing rules for type abstractions and applications:
  
  - Type Abstraction:

    \[
    \Delta[\alpha \leftarrow \star], \Gamma \vdash e : \tau \\
    \frac{}{\Delta, \Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau}
    \]
Typing Rules cont.

▶ And now the typing rules for type abstractions and applications:

▶ Type Abstraction:

\[
\frac{\Delta[\alpha \leftarrow \star], \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau}
\]

▶ Type Application:

\[
\frac{\Delta, \Gamma \vdash e_1 : \forall \alpha. \tau_1}{\Delta, \Gamma \vdash (e_1 \tau) : \tau_1[\tau / \alpha]}
\]
Typing Rules cont.

▶ And now the typing rules for type abstractions and applications:

▶ Type Abstraction:

\[
\Delta[\alpha \leftarrow \star], \Gamma \vdash e : \tau \\
\hline
\Delta, \Gamma \vdash \Lambda \alpha. e : \forall \alpha.\tau
\]

▶ Type Application:

\[
\Delta, \Gamma \vdash e_1 : \forall \alpha.\tau_1 \\
\hline
\Delta, \Gamma \vdash (e_1 \tau) : \tau_1[\tau/\alpha]
\]

▶ Typing rules for other constructs unchanged except that any rule of the form \( \Gamma \vdash e : \tau \) now becomes \( \Delta, \Gamma \vdash e : \tau \)
Type Derivation Example

Show the type derivation for the following expression:

$$(((\Lambda \alpha . \lambda x : \alpha . x) \text{Int}) \ 3)$$
Type Derivation Example

Show the type derivation for the following expression:

1. \(((\Lambda \alpha. \lambda x : \alpha. x) \text{Int}) \ 3)\)
2. \(\Lambda \alpha. \lambda x : \alpha \rightarrow \alpha. \lambda y : \alpha. (x \ y)\)
Type Derivation Example

Show the type derivation for the following expression:

1. \(((\Lambda \alpha. \lambda x : \alpha. x) \text{Int}) \; 3)\)
2. \(\Lambda \alpha. \lambda x : \alpha \rightarrow \alpha. \lambda y : \alpha. (x \; y)\)
3. \(\Lambda \alpha. \Lambda \beta. \lambda x : \alpha \rightarrow \beta. \lambda y : \alpha. (x \; y)\)
The polymorphic type system we looked at today is called System F.
The polymorphic type system we looked at today is called **System F**

**System F = simply-typed lambda calculus + polymorphism**
The polymorphic type system we looked at today is called System F.

System F = simply-typed lambda calculus + polymorphism

System F is sound: can prove progress and preservation.
Polymorphic Lambda Language

- The polymorphic type system we looked at today is called **System F**

- **System F** = simply-typed lambda calculus + polymorphism

- System F is **sound**: can prove progress and preservation

- Makes it possible to write generic functions that can work on arguments of any type
The polymorphic type system we looked at today is called **System F**

**System F** = simply-typed lambda calculus + polymorphism

**System F** is **sound**: can prove progress and preservation

Makes it possible to write generic functions that can work on arguments of any type

Furthermore, can easily extend it to allow polymorphic data structures
However, System F has one limitation.
Polymorphic Lambda Language Limitations

- However, System F has one limitation

- Sometimes we want to use operations that only make sense on some types, but not all types

Example: Operator + may be defined on Integers and Floats, but not vectors.
Polymorphic Lambda Language Limitations

- However, System F has one limitation
- Sometimes we want to use operations that only make sense on some types, but not all types
- **Example**: Operator + may be defined on Integers and Floats, but not vectors
Polymorphic Lambda Language Limitations

▶ However, System F has one limitation

▶ Sometimes we want to use operations that only make sense on some types, but not all types

▶ **Example**: Operator + may be defined on Integers and Floats, but not vectors

▶ Typing rules for System F don’t allow that: function definition allowed if the body type checks for *any* possible type.
Polymorphic Lambda Language Limitations

- However, System F has one limitation

- Sometimes we want to use operations that only make sense on some types, but not all types

- **Example:** Operator + may be defined on Integers and Floats, but not vectors

- Typing rules for System F don’t allow that: function definition allowed if the body type checks for any possible type.

- For this reason, real-world implementations of polymorphism do not stop here.
Polymorphism for Some Types

- **First Solution**: Evaluate type applications, then do the type checking!
Polymorphism for Some Types

- **First Solution:** Evaluate type applications, then do the type checking!

- Specifically, evaluate \(((\Lambda x : \alpha.e) \tau)\), then type check \(e[\tau/\alpha]\)
Polymorphism for Some Types

- **First Solution**: Evaluate type applications, then do the type checking!

- Specifically, evaluate \(((\Lambda x : \alpha. e) \tau), \text{ then type check } e[\tau/\alpha]\)

- **Example**: let \(x = \Lambda \alpha. \lambda y : \alpha. y + 1\) in \((x \text{ Int } 3)\) will not type check under our typing rules, but will type check now.
First Solution Trade Offs

- **Advantages:**
  - We allow more correct programs
  - Can write functions that are generic over a set of types

- **Disadvantages:**
  - Adding a new function call may mean your program no longer type checks!
  - Need to reanalyze function for every new call site, losing locality ⇒ longer compilation time
First Solution Trade Offs

- **Advantages:**
  - We allow more correct programs
  - Can write functions that are generic over a set of types
First Solution Trade Offs

▶ Advantages:
  ▶ We allow more correct programs
  ▶ Can write functions that are generic over a set of types

▶ Disadvantages:
First Solution Trade Offs

▶ Advantages:

▶ We allow more correct programs

▶ Can write functions that are generic over a set of types

▶ Disadvantages:

▶ Adding a new function call may mean your program no longer type checks!

▶ Need to reanalyze function for every new call site, losing locality ⇒ longer compilation time
Polymorphism in C++

Anyone knows a language that implements polymorphism with these properties?
Polymorphism in C++

Anyone knows a language that implements polymorphism with these properties?  C++
Polymorphism in C++

- Anyone knows a language that implements polymorphism with these properties? C++

- In C++, polymorphism achieved through templates:

  ```cpp
  template <class T> T foo(T arg) ...
  ```
Polymorphism in C++

- Anyone knows a language that implements polymorphism with these properties? C++

- In C++, polymorphism achieved through templates:
  
  ```cpp
  template <class T> T foo(T arg) ...
  ```

- Pro: Can write functions that are generic for a set of types
Polymorphism in C++

▶ Anyone knows a language that implements polymorphism with these properties? C++

▶ In C++, polymorphism achieved through templates:

```cpp
template <class T> T foo(T arg) ...
```

▶ Pro: Can write functions that are generic for a set of types

▶ Con: Really long compile times!
Polymorphism in C++

Anyone knows a language that implements polymorphism with these properties? **C++**

In C++, polymorphism achieved through **templates**:

```cpp
template <class T> T foo(T arg) ...
```

**Pro**: Can write functions that are generic for a set of types

**Con**: Really long compile times!

And new errors when instantiating template with a new type
Polymorphism in Java

- Java picked different strategy for allowing polymorphism for a set of types

  Idea: Qualify the type $\alpha$ as supporting some operations (i.e., implements some interface)

  Java syntax: `void drawAll(List<?> shapes)`

  Now, to require that $\alpha$ implements a interface, you write `void drawAll(List<? implements Shape> shapes)`
Polymorphism in Java

- Java picked different strategy for allowing polymorphism for a set of types

- **Idea:** Qualify the type $\alpha$ as supporting some operations (i.e., implements some interface)
Polymorphism in Java

- Java picked different strategy for allowing polymorphism for a set of types

- **Idea:** Qualify the type $\alpha$ as supporting some operations (i.e., implements some interface)

- Java syntax: `void drawAll(List<? implements Shape> shapes)` defines a function that takes lists with any type of element
Polymorphism in Java

- Java picked different strategy for allowing polymorphism for a set of types

- **Idea:** Qualify the type $\alpha$ as supporting some operations (i.e., implements some interface)

- **Java syntax:** `void drawAll(List<? implements Shape> shapes)` defines a function that takes lists with any type of element

- Now, to require that `?` implements a interface, you write `void drawAll(List<? implements Shape> shapes)`
Conclusion about Polymorphism

- Over last several years, polymorphism has gone main stream
Conclusion about Polymorphism

- Over last several years, polymorphism has gone main stream
- Many languages either substantially extend their treatment of polymorphism (C++) or added polymorphism (Java, C#)
Conclusion about Polymorphism

- Over last several years, polymorphism has gone main stream
- Many languages either substantially extend their treatment of polymorphism (C++) or added polymorphism (Java, C#)
- However, polymorphism always tends to be a difficult addition to any language.
Conclusion about Polymorphism

- Over last several years, polymorphism has gone main stream

- Many languages either substantially extend their treatment of polymorphism (C++) or added polymorphism (Java, C#)

- However, polymorphism always tends to be a difficult addition to any language.

- You either are already using it or will use it soon
Motivation for Type Inference

- So far when we studied typing, we always assumed that the programmer annotated some types.
Motivation for Type Inference

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

- **Example:** We declared types for let and lambda-bound variables
Motivation for Type Inference

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

- **Example**: We declared types for let and lambda-bound variables.

- Guaranteeing lack of run-time errors is nice, but type annotations can be cumbersome!
Motivation for Type Inference

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

- **Example:** We declared types for let and lambda-bound variables

- Guaranteeing lack of run-time errors is nice, but type annotations 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 benefits of static typing
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 benefits of 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:

\[
\begin{align*}
\Gamma \vdash i : Int & \quad \Gamma \vdash s : String \\
\Gamma \vdash id : \Gamma(id) \\
\Gamma \vdash S_1 : Int & \quad \Gamma \vdash S_1 : String \\
\Gamma \vdash S_2 : Int & \quad \Gamma \vdash S_2 : String \\
\Gamma \vdash S_1 + S_2 : Int & \quad \Gamma \vdash S_1 :: S_2 : String \\
\Gamma \vdash S_1 : \tau_1 & \quad \tau = \tau_1 \\
\Gamma[id \leftarrow \tau] \vdash S_2 : \tau_3 \\
\Gamma \vdash \text{let } id : \tau = S_1 \text{ in } S_2 : \tau_3 \\
\Gamma[x \leftarrow \tau_1] \vdash S_1 : \tau_2 \\
\Gamma \vdash \lambda x : \tau_1.S_1 : \tau_1 \rightarrow \tau_2 & \quad \Gamma \vdash S_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash S_2 : \tau_1 & \quad \Gamma \vdash (S_1 S_2) : \tau_2 \\
\end{align*}
\]
Type System

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

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

\[
\begin{align*}
\Gamma \vdash S_1 : \text{Int} & \quad \Gamma \vdash S_1 : \text{String} \\
\Gamma \vdash S_2 : \text{Int} & \quad \Gamma \vdash S_2 : \text{String} \\
\Gamma \vdash S_1 + S_2 : \text{Int} & \quad \Gamma \vdash S_1 :: S_2 : \text{String} \\
\Gamma \vdash S_1 : \tau_1 & \\
\tau = \tau_1 & \\
\Gamma[\text{id} \leftarrow \tau] \vdash S_2 : \tau_3 & \\
\Gamma \vdash \text{let } \text{id} : \tau = S_1 \text{ in } S_2 : \tau_3 & \\
\Gamma \vdash S_1 : \tau_1 \rightarrow \tau_2 & \\
\Gamma \vdash S_2 : \tau_1 & \\
\Gamma \vdash (S_1 S_2) : \tau_2 & \\
\end{align*}
\]
But, do we actually need these type annotations to infer the type of programs?
But, do we actually need these type annotations to infer the type of programs?

Consider the following example:

```plaintext
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:
  \[
  \text{let } f1 = \text{lambda } x.x+2 \text{ in ..}
  \]

- 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:

```plaintext
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:

```latex
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 $\text{Int} \rightarrow \text{Int}$.
Consider the following example:

```
let f2 = lambda x.lambda y.x+y in ..
```
Consider the following example:

```c
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:

```plaintext
let f2 = lambda x.lambdas 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:

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

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 \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)
Consider the following example:

```lambda
let f2 = lambda x. lambda y. x+1 in ..
```
Consider the following example:

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

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

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

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 \( \forall \alpha. \text{Int} \rightarrow \alpha \rightarrow \text{Int} \).
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. Therefore, the type of `g` must be `∀ α. Int → α`, but its return value can be anything. This means that the type of `f2` is `∀ α. (Int → α) → α`. 
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.

Therefore, the type of \( f2 \) is \( \forall \alpha . \text{Int} \to \alpha \), and its return value can be anything.

This means that the type of \( f2 \) is \( \forall \alpha . (\text{Int} \to \alpha) \to \alpha \).
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
Now, consider the following example:

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

Here, we know that function $f_2$ 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 $\forall \alpha. \text{Int} \to \alpha$, but its return value can be anything.
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 \( \forall \alpha. \text{Int} \rightarrow \alpha \), but its return value can be anything.

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

➤ **Goal of the rest of this lecture:** Develop an algorithm to compute the most general type for any expression
Goal of the rest of this lecture: Develop an algorithm to compute the most general type for any expression

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 to compute the most general type for any expression.

- 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:

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

- **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 $a$:

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

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

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

Observe that we have one additional precondition on the plus rule: The type variable `a` must be equal to `Int`
Type Variables Cont.

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

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

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

- Since $a = Int$, final type: $Int \rightarrow Int$
Type Variables in Typing Rules

- 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 fresh 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 fresh type variable $a$ for the type of $x$
  - Every time a rule uses the type of $x$, we use type variable $a$
Type Variables in Typing Rules

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

- We introduced a fresh type variable $a$ for the type of $x$.
- Every time a rule uses the type of $x$, we use type variable $a$.
- 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 $a = \text{Int}$. 
Type Variables in Typing Rules

▶ In this example, we dealt with not knowing the type of $x$ in the following way:

▶ We introduced a fresh type variable $a$ for the type of $x$

▶ Every time a rule uses the type of $x$, we use type variable $a$

▶ 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:

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

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

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

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

  \[
  \Gamma \vdash S_1 + S_2 : Int
  \]

- The lines marked in red are constraints.
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.

- 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:

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

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 concatenation:

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

The lines marked in red are again constraints.
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:

\[
\begin{align*}
\Gamma &\vdash S_1 : \tau_1 \\
\Gamma[\text{id} \gets a] &\vdash S_2 : \tau_2 \ (a \text{ fresh}) \\
a = \tau_1 \\
\hline
\Gamma &\vdash \text{let } id = S_1 \text{ in } S_2 : \tau_2
\end{align*}
\]
The Let Case

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

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

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:

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

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

Since \( S_1 \) has type \( \tau_1 \), add constraint \( a = \tau_1 \).
The Lambda Case

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

\[
\Gamma \left[ x \leftarrow a \right] \vdash S_1 : \tau \quad (a \text{ fresh})
\]

\[
\Gamma \vdash \lambda x. S_1 : a \rightarrow \tau
\]
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 \( x \).
The Lambda Case

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

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

- 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:

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

- 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 \(a\)
Now the only rule missing so far is application:

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

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:

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

\[
\begin{align*}
\text{identifier } x & \quad \Gamma(x) = a_1 \\
\Gamma[x ← a_1] ⊢ x : a_1 & \quad \text{integer } 2 \\
\Gamma[x ← a_1] ⊢ 2 : \text{Int} & \quad a_1 = \text{Int}, \text{Int} = \text{Int} \\
\Gamma[x ← a_1] ⊢ x + 2 : \text{Int} & \\
\Gamma ⊢ \lambda x. x + 2 : a_1 \rightarrow \text{Int}
\end{align*}
\]
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:

\[
\begin{align*}
\text{identifier } x & \quad \Gamma(x) = a_1 \\
\Gamma[x \leftarrow a_1] \vdash x : a_1 & \quad \text{integer } 2 \\
\Gamma[x \leftarrow a_1] \vdash 2 : \text{Int} & \quad a_1 = \text{Int}, \text{Int} = \text{Int} \\
\Gamma[x \leftarrow a_1] \vdash x + 2 : \text{Int} & \quad \Gamma \vdash \lambda x. x + 2 : a_1 \rightarrow \text{Int}
\end{align*}
\]

Type of expression: \( a_1 \rightarrow \text{Int} \) under constraints \( a_1 = \text{Int}, \text{Int} = \text{Int} \)
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:

  \[
  \begin{align*}
  \Gamma(x) &= a_1 \\
  \Gamma[x \leftarrow a_1] &\vdash x : a_1 \\
  \Gamma[x \leftarrow a_1] &\vdash 2 : \text{Int} \\
  \Gamma[x \leftarrow a_1] &\vdash x + 2 : \text{Int} \\
  \Gamma &\vdash \lambda x.x + 2 : a_1 \to \text{Int}
  \end{align*}
  \]

- Type of expression: \( a_1 \to \text{Int} \) under constraints \( a_1 = \text{Int}, \text{Int} = \text{Int} \)

- Since \( a_1 = \text{Int} \), inferred type: \( \text{Int} \to \text{Int} \)
Example 2

▶ What about the expression?

```
let f = lambda x. x + 1 in (f 2)
```
Example 2

- What about the expression?

```plaintext
let f = lambda x. x + 1 in (f 2)
```

- Type derivation:

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

- What about the expression?

\[
\text{let } f = \lambda x. x + 1 \text{ in } (f 2)
\]

- Type derivation:

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

\[
\Gamma \vdash \text{let } f = \lambda x.x \text{ in } f : a_2
\]

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

- What about the expression?

```latex
let f = \lambda x. x + 1 in (f 2)
```

- Type derivation:

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

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

- Hence, type is \( a_1 \rightarrow a_1 \)
Example 2 Cont

Final Type:  $a_1 \rightarrow a_1$, but what does this really mean?
Final Type: $a_1 \to a_1$, but what does this really mean?

Since no constraints on $a_1$, found a polymorphic type!
Example 2 Cont

- Final Type: $a_1 \rightarrow a_1$, but what does this really mean?
- Since no constraints on $a_1$, found a polymorphic type!
- Hence, the actual type is $\forall \alpha. \alpha \rightarrow \alpha$
Example 3

- Let’s look at the following expression "duck" + 7
Example 3

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

Type derivation:

\[ \Gamma \vdash \text{"duck" : String} \]
\[ \Gamma \vdash 7 : Int \]
\[ String = Int, Int = Int \]
\[ \Gamma \vdash \text{"duck" + 7 : Int} \]
Example 3

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

- Type derivation:

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

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

- Type derivation:

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

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

- These constraints are unsatisfiable!
Example 3

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

- Type derivation:

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

- 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:
Observe that we have split the problem of type inference into two separate problems:

1. **Constraint Inference**: Apply typing rules to compute the type in terms of type variables and derive type constraints.
Observe that we have split the problem of type inference into two separate problems:

1. **Constraint Inference**: Apply typing rules to compute the type in terms of type variables and derive type constraints

2. **Constraint Solving**: 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**: Apply typing rules to compute the type in terms of type variables and derive type constraints
  2. **Constraint Solving**: 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 typed in step 2.
Conclusion

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

Next time: How to efficiently solve the type constraints