

# Lecture 13: Polymorphism

Yu Feng  
Winter 2023

# Recap

- Last time we saw that we can build a static type system that prevents many run-time errors
- **Examples:** Adding ints and strings, applying a non-lambda term, ...
- But even in a sound type system we will prohibit some programs that would never have any run-time problems
- **Today:** How to extend static type systems to allow polymorphism

# Motivation

- Consider the following function in the untyped lambda language: `lambda x.x`
- Here, the following program is well-defined: `(lambda x.x 3)`
- But so is the following program: `(lambda x.x "duck")`
- And the following program: `(lambda x.x (lambda y.y*2))`
- This function can work on many (in this case, all) types!

# A Simple Type System

- How would you write `lambda x.x` in the **typed** lambda language?
- Here, types forces us to over-specialize the contexts in which this function works
- Type systems that force us to fully specify all types are known as **monomorphic** type systems

$$\begin{aligned} S &\rightarrow \text{integer} \mid \text{string} \mid \text{identifier} \\ &\quad \mid S_1 + S_2 \mid S_1 :: S_2 \\ &\quad \mid \text{let } id : \tau = S_1 \text{ in } S_2 \\ &\quad \mid \lambda x : \tau. S_1 \\ &\quad \mid (S_1 \ S_2) \\ \tau &\rightarrow \text{Int} \mid \text{String} \mid \tau_1 \rightarrow \tau_2 \end{aligned}$$

# Monomorphic Type Systems

- This problem usually becomes especially painful when implementing data structures
- You end up with a vector of Ints, Strings, Foo, ...
- Also quite common with numeric code to multiple matrices etc.
- However, most programmers experience the problem as **users** of library code, not so often as writers

# Solutions

- **First Solution:** Duplicate function for each type used
- Makes code large and hard to maintain
- Bugs need to be fixed in many places
- Every time there is one more type, you have to copy and paste again
- **Terrible Strategy**, still surprisingly common
- **Slogan:** Who needs polymorphism if we have copy and paste?

# Solutions

- **Second Solution:** Escape the type system
- In C, this means using a void\*
- In Java, this casts everything to Object
- But now we are back to run-time errors!

# Solutions

- **Second Solution:** Escape the type system
- In C, this means using a void\*
- In Java, this casts everything to Object
- But now we are back to run-time errors!



# Polymorphic Types

- So far, in our type system we only have **type constants**
- Examples: `Int`, `String`, `Int → Int`,...
- **Big Idea:** Introduce type variables that can range over any type

# Polymorphic Types

- Specifically, add the following **type abstraction** to our language:  
 $\Lambda\alpha.e$
- Think of this term as function that takes a type and substitute all occurrences of type  $\alpha$  in expression  $e$
- **Example:** Consider  $((\Lambda\alpha.\lambda x:\alpha.x) \text{ Int})$
- This evaluates to  $\lambda x:\text{Int}.x$

# Polymorphic Types

- But what is the type of an expression such as  $(\Lambda\alpha.\lambda x:\alpha.x)$ ?
- We will write the type of  $\Lambda\alpha.e$  where  $e$  evaluates to type  $\tau$  as  $\forall\alpha.\tau$
- **Intuition:** This type holds for all instantiations of the type variable  $\alpha$
- **Side Note:** It is no accident that this type starts to look like a logic formula
- **Curry-Howard Isomorphism** shows fundamental equivalence between types and logic formulas

# Polymorphic Lambda Language

$$\begin{aligned}
 S &\rightarrow \text{integer} \mid \text{string} \mid \text{identifier} \\
 &\quad \mid S_1 + S_2 \mid S_1 :: S_2 \\
 &\quad \mid \text{let } id : \tau = S_1 \text{ in } S_2 \\
 &\quad \mid \lambda x : \tau. S_1 \\
 &\quad \mid \Lambda \alpha. S_1 \\
 &\quad \mid (S_1 \ S_2) \mid (S_1 \ \tau) \\
 \tau &\rightarrow Int \mid String \mid \tau_1 \rightarrow \tau_2 \mid \alpha
 \end{aligned}$$

- Operational Semantics for  $\Lambda \alpha. S_1$

$$\overline{E \vdash \Lambda \alpha. S_1 : \Lambda \alpha. S_1}$$

- Operational Semantics for type application:

$$\frac{
 \begin{array}{l}
 E \vdash S_1 : \Lambda \alpha. e_1 \\
 E \vdash e_1[\tau/\alpha] : e_2
 \end{array}
 }{
 E \vdash (S_1 \ \tau) : e_2
 }$$

# Java Polymorphism

- Java syntax: *public void drawAll(List<?> shapes)*  
defines a function that takes lists with any type of element
- Observe how this is exactly like polymorphic lambda language, just different syntax
- Now, to require that ? implements an interface, you write *public void drawAll(List<? implements Shape> shapes)*

# Conclusion

- Over the last few 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