CS345H: Programming Languages

Lecture 11: Polymorphism

Thomas Dillig

Introduction

- Last time we saw that we can build a static type system that prevents many run-time errors

Introduction

- 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, ...

Introduction

- 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, ...
- We also discussed the two key properties of any type system: Preservation and Progress

Introduction

- 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, ...
- We also discussed the two key properties of any type system: Preservation and Progress
- But even in a sound type system we will prohibit some programs that would never have any run-time problems

Introduction

- 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, ...
- We also discussed the two key properties of any type system: Preservation and Progress
- 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`

Motivation

- Consider the following function in the untyped lambda language: `lambda x.x`
- Here, the following program is well-defined: `(lambda x.x 3)`

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")`

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))`

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!

Recall the Typed Lambda Language

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

Recall the Typed Lambda Language

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

- How would you write `lambda x.x` in the typed lambda language?

Recall the Typed Lambda Language

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

- 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

Recall the Typed Lambda Language

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

- 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

Monomorphic Type Systems

- This problem usually becomes especially painful when implementing data structures

Monomorphic Type Systems

- This problem usually becomes especially painful when implementing data structures
- You end up with a vector of Ints, Strings, Foo, ...

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.

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

- This problem has been observed for a long time!

Solutions

- This problem has been observed for a long time!
- In fact, John Backus of FORTRAN fame pointed this problem out in the first FORTRAN manual

Solutions

- This problem has been observed for a long time!
- In fact, John Backus of FORTRAN fame pointed this problem out in the first FORTRAN manual
- But he did not attempt to solve it

Solutions

- First Solution: Duplicate function for each type used

Solutions

- First Solution: Duplicate function for each type used
  - Makes code large and hard to maintain

Solutions

- First Solution: Duplicate function for each type used
  - Makes code large and hard to maintain
  - Bugs need to be fixed in many places

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

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

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 Cont.

- Second Solution: Escape the type system

Solutions Cont.

- Second Solution: Escape the type system
- In C, this means using a `void*`

Solutions Cont.

- Second Solution: Escape the type system
- In C, this means using a `void*`
- In Java, this casts everything to `Object`

Solutions Cont.

- 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

- Fortunately, it is possible to allow polymorphism in types

Polymorphic Types

- Fortunately, it is possible to allow polymorphism in types
- This will mean that we can write a function, such as `lambda x.x` that will type correctly and still have all the benefits from a sound type system

Polymorphic Types

- Fortunately, it is possible to allow polymorphism in types
- This will mean that we can write a function, such as `lambda x.x` that will type correctly and still have all the benefits from a sound type system
- We can have the cake and eat it!

Polymorphic Types

- Fortunately, it is possible to allow polymorphism in types
- This will mean that we can write a function, such as `lambda x.x` that will type correctly and still have all the benefits from a sound type system
- We can have the cake and eat it!
- This used to be mostly of academic interest, but has recently become mainstream in most languages (Java generics)

Polymorphic Types

- So far, in our type system we only have type constants

Polymorphic Types

- So far, in our type system we only have type constants
- Examples: Int, String, $Int \rightarrow Int$,...

Polymorphic Types

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

Polymorphic Types

- Specifically, add the following type abstraction to our language: $\Lambda\alpha.e$

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 α in expression e

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 α in expression e
- Example: Consider $((\Lambda\alpha.\lambda\ x{:}\alpha.x)Int)$

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 α in expression e
- Example: Consider $((\Lambda\alpha.\lambda\ x{:}\alpha.x)Int)$
- This evaluates to $\lambda\ x{:}Int.x$

Polymorphic Types Cont.

- But what is the type of an expression such as $(\Lambda\alpha.\lambda\ x{:}\alpha.x)$?

Polymorphic Types Cont.

- 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 τ as $\forall\alpha.\tau$

Polymorphic Types Cont.

- 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 τ as $\forall\alpha.\tau$
- Intuition: This type holds for all instantiations of the type variable α

Polymorphic Types Cont.

- 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 τ as $\forall\alpha.\tau$
- Intuition: This type holds for all instantiations of the type variable α
- Side Note: It is no accident that this type starts to look like a logic formula

Polymorphic Types Cont.

- 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 τ as $\forall\alpha.\tau$
- Intuition: This type holds for all instantiations of the type variable α
- 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

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

Polymorphic Lambda Language

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

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

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

Polymorphic Lambda Language

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

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

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

- Operational Semantics for type application:

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

Typing Rules Preliminaries

- Before we can design typing rules for our polymorphic lambda language, we have one problem

Typing Rules Preliminaries

- Before we can design typing rules for our polymorphic lambda language, we have one problem
- Consider the expression `let x:`α` = ...`

Typing Rules Preliminaries

- Before we can design typing rules for our polymorphic lambda language, we have one problem
- Consider the expression `let x:`α` = ...`
- Here, we don't want type checking to succeed if α is not bound by a type abstraction $\Lambda\alpha$

Typing Rules Preliminaries

- Before we can design typing rules for our polymorphic lambda language, we have one problem
- Consider the expression `let x:`α` = ...`
- Here, we don't want type checking to succeed if α is not bound by a type abstraction $\Lambda\alpha$
- Just like we use environment Γ to check that identifiers are used before they are defined, we need an additional environment Δ to track that all type variables α are bound

Typing Rules Preliminaries

- But type variables don't map to one type. We will use $\star$ to donate any well-formed type

Typing Rules Preliminaries

- But type variables don't map to one type. We will use $\star$ to donate any well-formed type
- Signature of Δ: $\alpha \mapsto \star$

Typing Rules Preliminaries

- But type variables don't map to one type. We will use $\star$ to donate any well-formed type
- Signature of Δ: $\alpha \mapsto \star$
- We will need a judgment $\Delta \vdash \tau : \star$ asserting that type τ is well-formed.

Typing Rules Preliminaries

- But type variables don't map to one type. We will use $\star$ to donate any well-formed type
- Signature of Δ: $\alpha \mapsto \star$
- We will need a judgment $\Delta \vdash \tau : \star$ asserting that type τ is well-formed.
- Intuitively, type τ is well-formed if all free variables in τ are in Δ

Well-formedness Rules

- Let's give rules for this judgment:

Well-formedness Rules

- Let's give rules for this judgment:
  - Base case 1:

$$\frac{}{\Delta \vdash Int : \star} \qquad \frac{}{\Delta \vdash String : \star}$$

Well-formedness Rules

- Let's give rules for this judgment:
  - Base case 1:

$$\frac{}{\Delta \vdash Int : \star} \qquad \frac{}{\Delta \vdash String : \star}$$

  - Base case 2:

$$\frac{}{\Delta \vdash \alpha : \Delta(\alpha)}$$

Well-formedness Rules Cont.

- On to the inductive rules:

Well-formedness Rules Cont.

- On to the inductive rules:
  - Inductive Case 1:

$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \rightarrow \tau_2 : \star}$$

Well-formedness Rules Cont.

- On to the inductive rules:
  - Inductive Case 1:

$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \to \tau_2 : \star}$$

  - 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:

$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \rightarrow \tau_2 : \star}$$

  - Inductive Case 2:

$$\frac{\Delta[\alpha \leftarrow \star] \vdash \tau : \star}{\Delta \vdash \forall \alpha . \tau : \star}$$

- All this says is that if $\Delta \vdash \tau : \star$ holds, type τ has no free variables

Typing Rules

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

Typing Rules

- Let's look at the typing rules affected by type variables:
  - Function definition:

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 : \star \\ \Delta, \Gamma[x \leftarrow \tau_1] \vdash e : \tau_2\end{array}}{\Delta, \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \rightarrow \tau_2}$$

Typing Rules

- Let's look at the typing rules affected by type variables:
  - Function definition:

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 : \star \\ \Delta, \Gamma[x \leftarrow \tau_1] \vdash e : \tau_2\end{array}}{\Delta, \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \rightarrow \tau_2}$$

Typing Rules

- Let's look at the typing rules affected by type variables:
  - Function definition:

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 : \star \\ \Delta, \Gamma[x \leftarrow \tau_1] \vdash e : \tau_2\end{array}}{\Delta, \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \rightarrow \tau_2}$$

  - 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:

$$\frac{\begin{array}{c}\Delta \vdash \tau_1 : \star \\ \Delta, \Gamma[x \leftarrow \tau_1] \vdash e : \tau_2\end{array}}{\Delta, \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \rightarrow \tau_2}$$

  - Observe that there are two different kinds of judgments here!

  - Type Abstraction Definition

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

Typing Rules

- And now the typing rules for applications:

Typing Rules

- And now the typing rules for applications:
  - Value Application:

$$\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta, \Gamma \vdash e_2 : \tau_1}{\Delta, \Gamma \vdash (e_1\ e_2) : \tau_2}$$

Typing Rules

- And now the typing rules for applications:
  - Value Application:

$$\frac{\Delta, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Delta, \Gamma \vdash e_2 : \tau_1}{\Delta, \Gamma \vdash (e_1 \ e_2) : \tau_2}$$

  - Type Application:

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

Typing Rules

- But what about the typing rules for plus and concatenation?

Typing Rules

- But what about the typing rules for plus and concatenation?
- Unchanged. These are only defined for Strings and Ints, not type variables

Typing Rules

- But what about the typing rules for plus and concatenation?
- Unchanged. These are only defined for Strings and Ints, not type variables
- In the lambda language this makes sense since there is no point in being polymorphic with respect to monomorphic operators

Polymorphic Lambda Language

- It is possible (and pretty straightforward) to prove that adding polymorphism preserves progress and preservation

Polymorphic Lambda Language

- It is possible (and pretty straightforward) to prove that adding polymorphism preserves progress and preservation
- But we won't do this in class today

Polymorphic Lambda Language

- It is possible (and pretty straightforward) to prove that adding polymorphism preserves progress and preservation
- But we won't do this in class today
- Enriching lambda calculus with types and polymorphism (but no let bindings) is also known as System F.

Polymorphic Lambda Language

- Our new polymorphic types worked great for giving types in the lambda language.

Polymorphic Lambda Language

- Our new polymorphic types worked great for giving types in the lambda language.
- But pretty much the only polymorphic functions we could write are variations of $\lambda x.x$!

Polymorphic Lambda Language

- Our new polymorphic types worked great for giving types in the lambda language.
- But pretty much the only polymorphic functions we could write are variations of $\lambda x.x$!
- Fortunately, if we also allow lists (like L), this kind of polymorphism still works and is very useful

Polymorphic Lambda Language

- Our new polymorphic types worked great for giving types in the lambda language.
- But pretty much the only polymorphic functions we could write are variations of $\lambda x.x$!
- Fortunately, if we also allow lists (like L), this kind of polymorphism still works and is very useful
- Typical use: Data structures

Polymorphic Lambda Language Limitations

- However, sometimes we have operations that only make sense on some types, but not all types

Polymorphic Lambda Language Limitations

- However, sometimes we have 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, sometimes we have operations that only make sense on some types, but not all types
- Example: Operator $+$ may be defined on Integers and Floats, but not vectors
- The typing rules we currently gave do not allow that. A function definition will only type check if the body type checks for any possible type.

Polymorphic Lambda Language Limitations

- However, sometimes we have operations that only make sense on some types, but not all types
- Example: Operator $+$ may be defined on Integers and Floats, but not vectors
- The typing rules we currently gave do not allow that. A function definition will only type check if the body type checks for any possible type.
- Type checking universal types for all possible instantiations is known as first-order semantics.

Polymorphic Lambda Language Limitations

- However, sometimes we have operations that only make sense on some types, but not all types
- Example: Operator $+$ may be defined on Integers and Floats, but not vectors
- The typing rules we currently gave do not allow that. A function definition will only type check if the body type checks for any possible type.
- Type checking universal types for all possible instantiations is known as first-order semantics.
- For this reason, real-world implementations of polymorphism do not stop here.

Polymorphism for Some Types

- First Solution: Only type check function definitions for the types that they are instantiated with!

Polymorphism for Some Types

- First Solution: Only type check function definitions for the types that they are instantiated with!
- Example: let $x = \Delta\alpha.\lambda y : \alpha.y + 1$ in $(x\ Int\ 3)$ will not type check under our typing rules, but will type check now.

Polymorphism for Some Types

- First Solution: Only type check function definitions for the types that they are instantiated with!
- Example: let $x = \Delta\alpha.\lambda y : \alpha.y + 1$ in $(x\ Int\ 3)$ will not type check under our typing rules, but will type check now.
- For this, we need another environment in our typing rules that "carries" the body of all functions to the application sites to be type checked at every application with the current type

Polymorphism for Some Types

- First Solution: Only type check function definitions for the types that they are instantiated with!
- Example: let $x = \Delta\alpha.\lambda y : \alpha.y + 1$ in $(x\ Int\ 3)$ will not type check under our typing rules, but will type check now.
- For this, we need another environment in our typing rules that "carries" the body of all functions to the application sites to be type checked at every application with the current type
- This is known as Herbrand semantics

First Solution Trade Offs

- Advantages:

First Solution Trade Offs

- Advantages:
  - We allow more correct programs

First Solution Trade Offs

- Advantages:
  - We allow more correct programs
  - We don't report errors that can never happen

First Solution Trade Offs

- Advantages:
  - We allow more correct programs
  - We don't report errors that can never happen
  - We allow polymorphism to be used in many more cases

First Solution Trade Offs

- Advantages:
  - We allow more correct programs
  - We don't report errors that can never happen
  - We allow polymorphism to be used in many more cases
  - Easy to implement as just cloning the code for each type

First Solution Trade Offs

- Disadvantages:

First Solution Trade Offs

- Disadvantages:
  - Adding a new application (call) may mean your program no longer type checks!

First Solution Trade Offs

- Disadvantages:
  - Adding a new application (call) may mean your program no longer type checks!
  - Need to reanalyze function for every new call site, losing locality

First Solution Trade Offs

- Disadvantages:
  - Adding a new application (call) may mean your program no longer type checks!
  - Need to reanalyze function for every new call site, losing locality
  - If generating code, this may mean recompilation of library for each new client!

Polymorphism by Code Cloning

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

Polymorphism by Code Cloning

- Anyone knows a language that implements polymorphism with these properties?
- C++ (who else)

Polymorphism by Code Cloning

- Anyone knows a language that implements polymorphism with these properties?
- C++ (who else)
- Still quite effective and potentially extremely efficient.

Polymorphism by Code Cloning

- Anyone knows a language that implements polymorphism with these properties?
- C++ (who else)
- Still quite effective and potentially extremely efficient.
- But the price is terrible compile times.

Polymorphism by Code Cloning

- Anyone knows a language that implements polymorphism with these properties?
- C++ (who else)
- Still quite effective and potentially extremely efficient.
- But the price is terrible compile times.
- And new errors when instantiating a template with a new type

Polymorphism for Some Types

- Java picked a different strategy when adding support for generics called type classes

Polymorphism for Some Types

- Java picked a different strategy when adding support for generics called type classes
- Idea: Qualify the type α as supporting some operations (being part of a type class)

Polymorphism for Some Types

- Java picked a different strategy when adding support for generics called type classes
- Idea: Qualify the type α as supporting some operations (being part of a type class)
- In Java, this is done by requiring that a polymorphic type implements some interface

Java Polymorphism

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

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

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 a interface, you write `public void drawAll(List<? implements Shape> shapes)`

Conclusion

- Over the last few years, polymorphism has gone main stream

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#)

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.

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