Introduction to Type Systems

Shan Shan Huang

Before we begin...

- Ask questions – this is the only way to go in depth.
- If you are really interested, try Cardelli and Wegner's *“On Understanding Types, Data Abstraction, and Polymorphism”*, (ACM Computing Surveys 17(4), Dec. 1985)
  - classical survey on types.
  - Yannis thinks it takes a summer to read...
  - I've been reading it for 5 years.

What is a Type?

- Defines a set of values
  - defines the allowed behavior of these values
  - a value can have more than one type!
- Examples of types
  - simple types: int, boolean, float, etc.
  - composite types: records, classes, functions

```
int power(int a, int exp) : (int, int) -> int
```

  - parametric types or type templates: arrays

```
int[]     : array<int>
String[] : array<String>
```

Typed vs. Untyped

- Typed languages
  - Statically typed: Java, C, etc.
  - Dynamically typed: Python, Javascript, etc.
- Untyped languages?
- For the remainder of the lecture: Typed <=> Statically Typed.

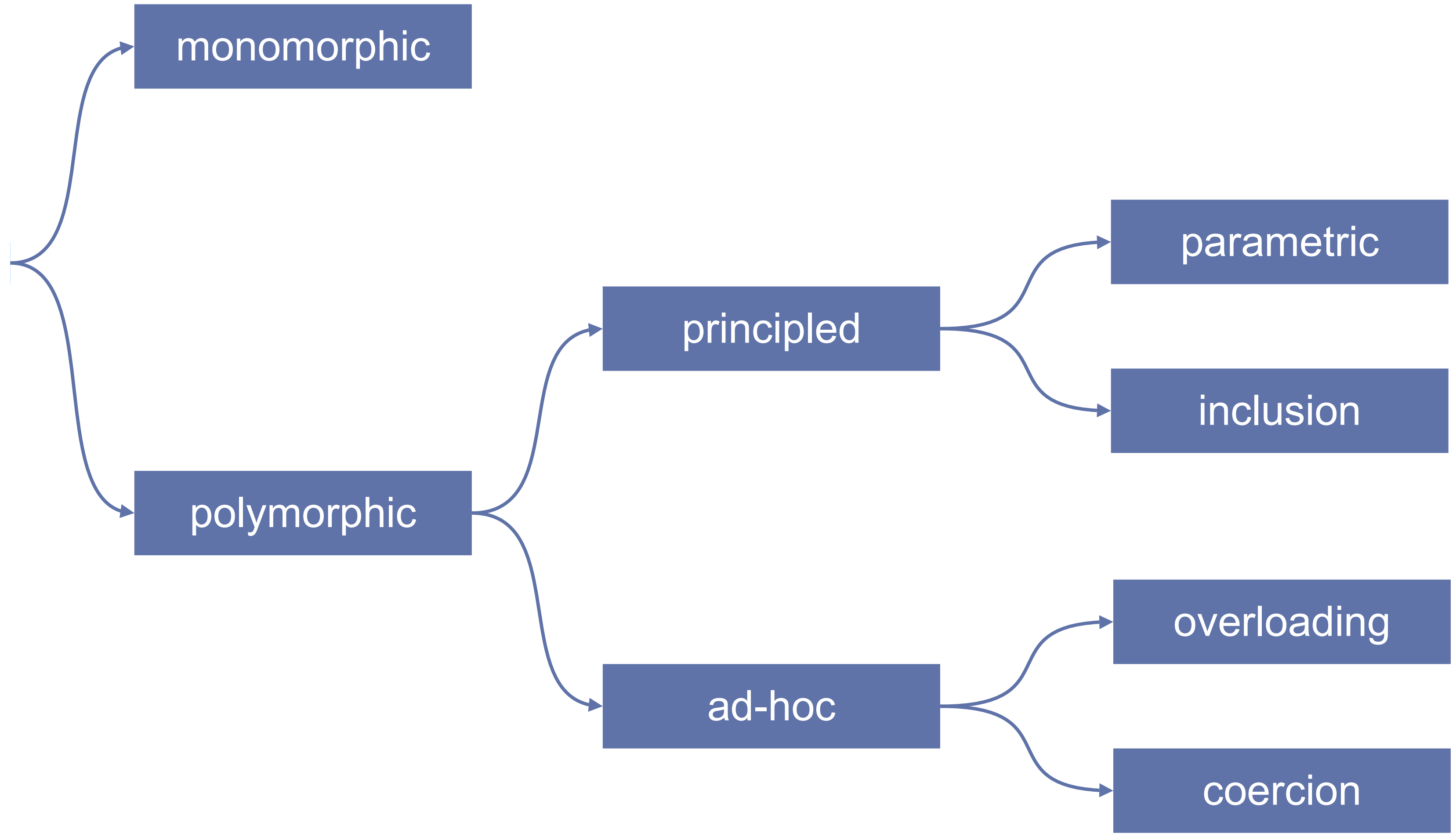
Why use Types?

- Types define static properties of values
  - Catch errors -- job of the type checker
    - Why catch errors statically if there are runtime checks anyway?
    - What are some errors that cannot be caught statically?
      - Array index out of bounds?
      - Divide by zero?
      - Dereferencing a null object?
  - Optimization

Properties of Type Systems

- What kind of types can a user define?
  - If we consider defined types as “constants” in the type system, can there be variables?
  - A type system is its own language!

Type Systems Anatomy



Monomorphic Types

- Each value/variable can have only one type.

```
struct ListNode {
   int data;
   struct ListNode *next;
}
```

- Values: held by variables declared as **`ListNode`**.
- Behavior: can reference **`data`** and **`next`**.
- Every symbol used to define the type is a constant in the **type system**
- Recursion is allowed.

Parametric Polymorphism

- Let's allow **type variables** in the definition of types.

```
interface List<E> {
   boolean add    (E       o);
   E       get    (int index);
}
```

- Possible meanings?
  - **`List`** may not be a type, but a type *template*.
  - **`List`** may be the set objects that accept **`add`** with argument of ***any*** type **`E`** (universal quantification)
  - **`List`** may be the set of objects that accept **`get`** that returns an object of ***some*** type **`E`** (existential quantification)

Type Templates

```
interface List<E> {
   boolean add   (E      o);
   E       get   (int index);
}
```

- Type templates are ***not*** types
  - Litmus test: can you use **`List`** as the type of a method argument?

```
void foo(List l) { ... }
```

  - For type templates, the answer is NO
  - Instead:

```
void foo(List<Integer> l) { ... }
```

Universal Quantification

```
interface List<forany E> {
   boolean add   (E       o);
   E       get   (int index);
}
```

```
List<Integer> intList;
intList.add(new Integer(3));
Integer i = intList.get(0);
```

```
List<String> strList;
strList.add(“foo”);
String s = strList.get(0);
```

Universal Quantification: Testing Your Understanding

```
interface List2 {
   <forany E> boolean add   (E      o);
   <forany E> E       get   (int index);
}
```

```
List2 lst = ... ;
lst.add("foo");
lst.add(new Integer(3));
lst.add(lst);
... "bar" == lst.get(0) ... ;
... lst == lst.get(2) ... ;
```

- Universally quantified return type violates soundness, unless there are values that belong to all types! (e.g. **`null`**)

Existential Quantification

```
interface ValueContainer<exists A> {
   A   value;
   int valueToInt(A a);
}
```

- You can only manipulate type **A** through interface **ValueContainer**.

```
ValueContainer v = ...;
int i = v.valueToInt(v.value());
```

Existential Quantification: Testing Your Understanding

```
interface ValueContainer<exists A> {
   A   value;
   int valueToInt(A a);
}
```

```
class VC3 {
   String value;
   int    valueToInt(List a) { ... }
}
```

- Does the above class **`implement ValueContainer`**?

Why use Existential Types?

- Existential types are for data-hiding.

```
interface Complex<exists R> {
   R r; // representation
   R makeComplex(float r1, float r2);
   float getReal(R r);
   float getImaginary(R r);
}
```

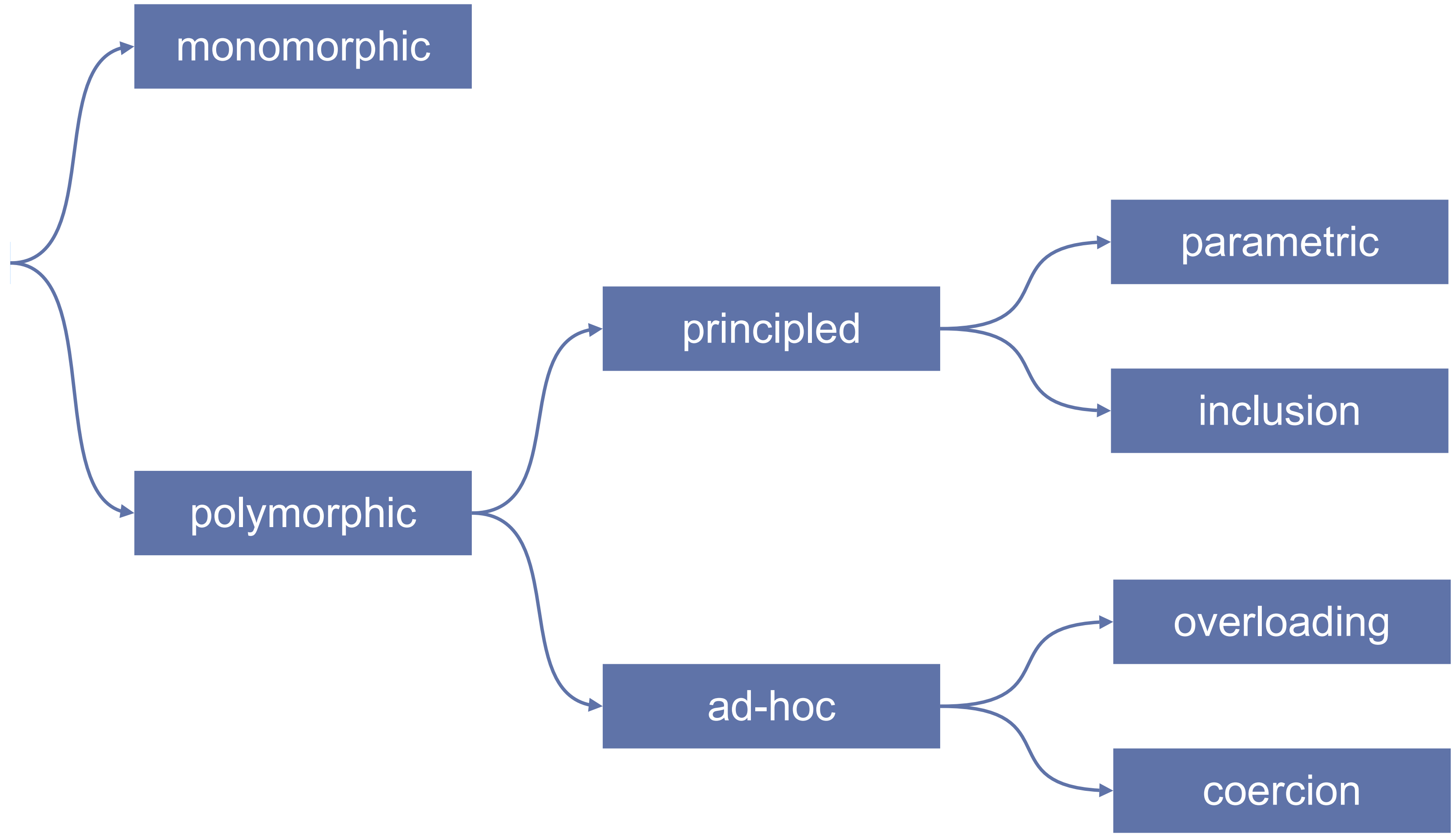
- Hide the actual implementation of the complex number **R**, but allows manipulation.
- In OO languages, we don't use mechanisms like existential types. Comparison to follow after discussion of subtyping.

Existential Together with Universal

```
interface List<forany E> { ... }
interface ValueContainer<exists A> { ... }
List<ValueContainer<exists A>> valueList;
```

- A heterogeneous **`List`** of **`ValueContainers`**, containing values of different types.

Type Systems Anatomy



Subtyping

- Values of a subtype ⊆ values of its the supertype.
- What does that mean for method signatures?

```
interface I1 {
   Animal foo(Dog d);
}

interface I2 extends I1 {
   Dog    foo(Animal d);
}

interface I3 extends I1 {
   Object foo(PrettyDog d);
}

interface I4 extends I1 {
   Dog    foo(Dog d);
}
```

```
Animal a;
Dog dog;
I1 i1Obj;

I2 i2Obj = ...;
i1Obj = i2Obj;
a = i1Obj.foo(dog);

I3 i3Obj = ...;
i1Obj = i3Obj;
a = i1Obj.foo(dog);

I4 i4Obj = ...;
i1Obj = i4Obj;
a = i1Obj.foo(dog);
```

Covariance and Contravariance

- Method argument: ***contravariant*** position (reverses ordering)
  - argument in subtype needs to be the ***supertype*** of argument in supertype
- Method return type: ***covariant*** position (preserves ordering)
  - return type in subtype needs to be a ***subtype*** of return type in supertype

```
interface I1 {
   Animal foo(Dog d);
}

interface I2 extends I1 {
   Dog    foo(Animal d);
}
```

```
interface I3 extends I1 {
   Object foo(PrettyDog d);
}

interface I4 extends I1 {
   Dog    foo(Dog d);
}
```

Subtyping vs. Parametric Polymorphism (1)

- Subtyping vs. Universal Types:
  - Subtyping allows (homogeneous) data structures by using common supertype (e.g. **`Object`**) as element type.
    - But when elements are extracted from data structure, they need to be casted back to their type – not statically type safe!
  - Universal types allow homogeneous data structures safely:

```
interface List<forany E> {
   boolean add   (E       o);
   E        get   (int index);
}
```

Subtyping vs. Parametric Polymorphism (2)

- Subtyping vs. Existential Types:

```
interface Complex<exists R> {
   R r; // representation
   R makeComplex(float r1, float r2);
   float getReal(R r);
   float getImaginary(R r);
}
```

  - Use common supertype for **`R`**
  - classes provide implementation and hide details of representation

Bounded Quantification

- Constrain the type variable using subtyping

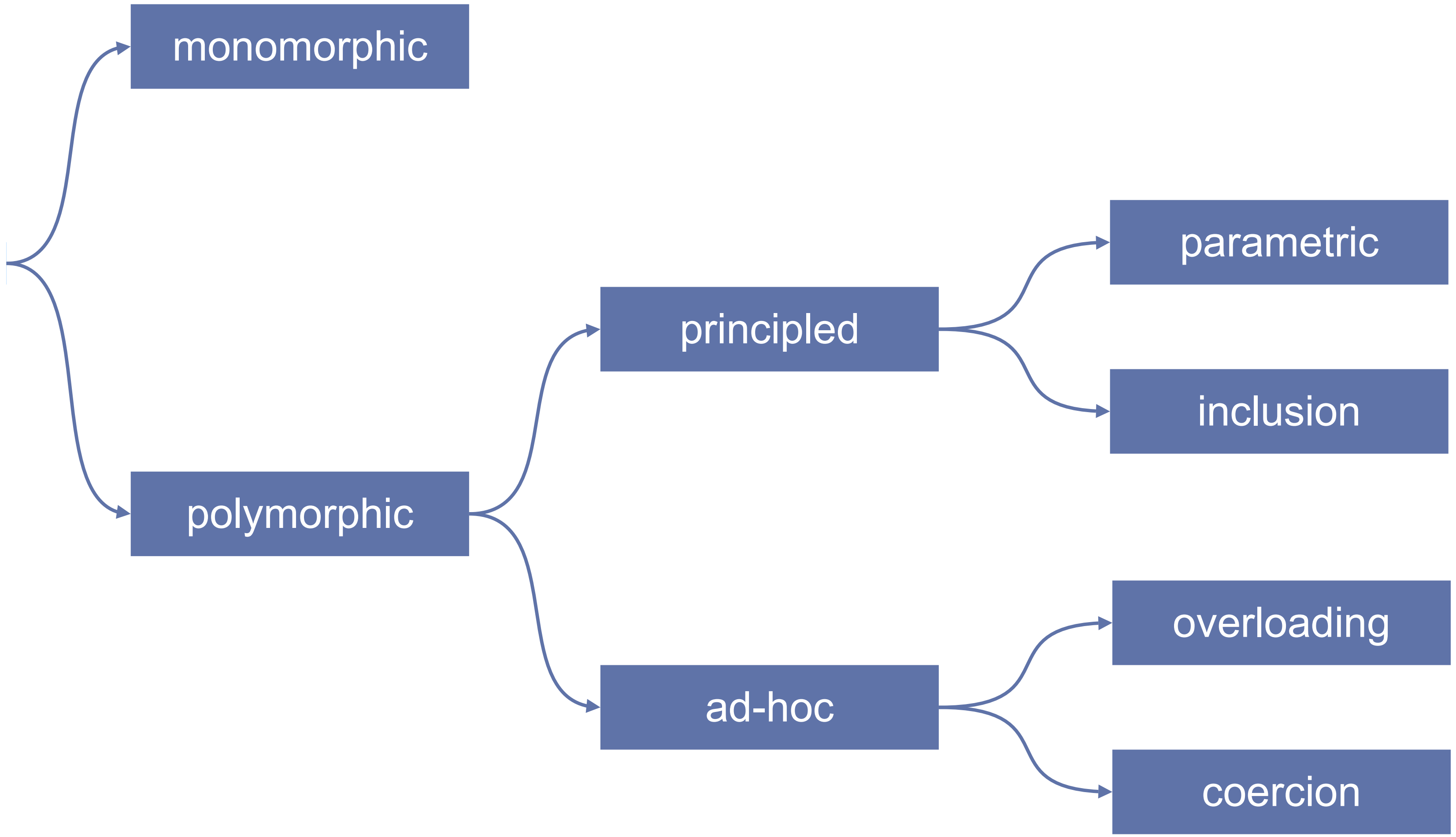
```
interface List<E extends Number> {
   boolean add   (E        o);
   E        get   (int index);
}
```

- F-bounded polymorphism: bound can be parameterized by a type variable.

```
interface Comparable<A> {
   int compareTo(A that);
}

interface List<E extends Comparable<E>> {
   boolean add   (E        o);
   E        get   (int index);
}
```

Type Systems Anatomy



Overloading & Coercion

- Ways to make function types somewhat “polymorphic”
- Overloading

```
interface List<E> {
   boolean add    (E o);
   boolean add    (int index, E o);
}
```

  - `add: E -> boolean, (E, int) -> boolean`
- Coercion

```
float divide(float i, float j) { ... }
```

`(float, float) -> float, (float, int) -> float`
`(int, float) -> float, (int, int) -> float`