Polymorphism

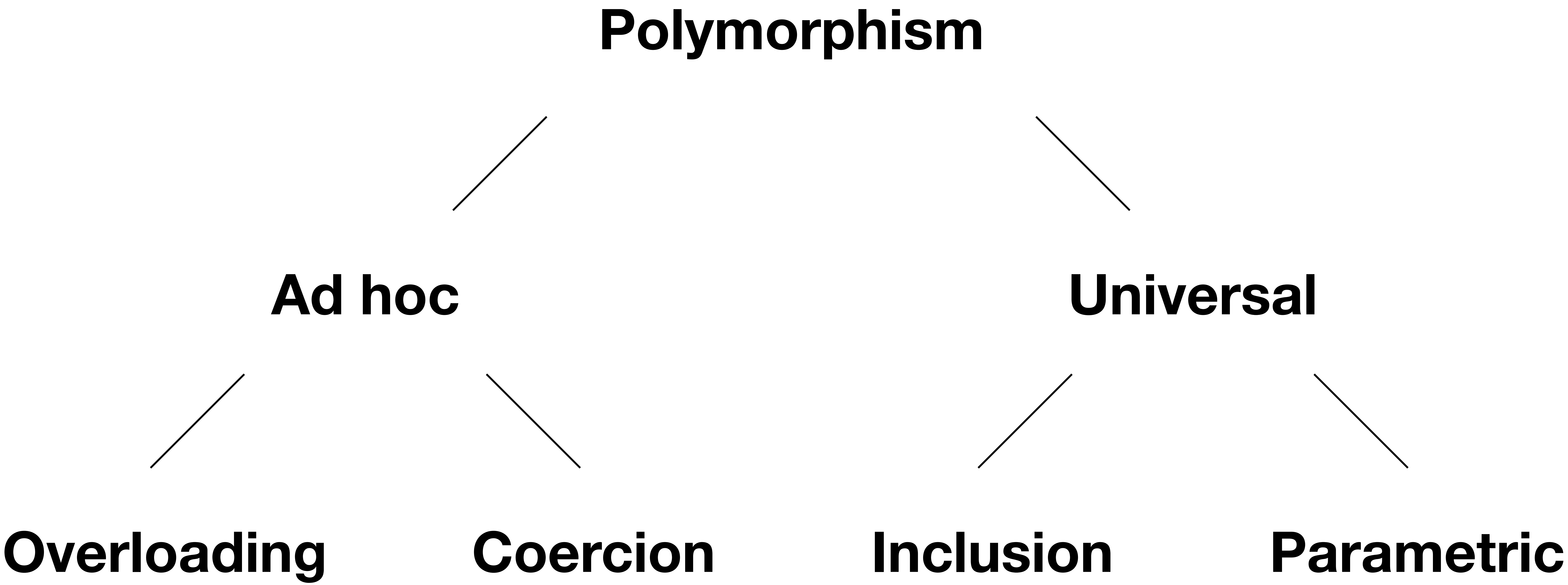
Monomorphism

- In monomorphic languages everything has exactly one type
- Examples: Pascal, C
- Our type checker so far is monomorphic

Polymorphism

- Functions are *polymorphic* if they appear to have more than one type
- Examples:
    - Mathematical operators in most programming languages: +, *, -, etc.
    - Overloaded functions in C++
    - Dynamically inherited methods in OO languages
      (e.g., virtual methods in C++, all methods in Java)
    - List length in Racket, Haskell

Types of Polymorphism



Polymorphism

Ad hoc

Universal

Overloading

Coercion

Inclusion

Parametric

Ad Hoc Polymorphism

- Appearance of polymorphism but really just thrown in here and there
- Depends on what variations the implementer decides to do
- Example (C++): What's the type signature for "+"?

```
3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0
```

- Two common options:
  - Overloading: multiple functions, one for each type signature
  - Coercion: some types can be converted to others to match type signature
- In both cases, each case of input or each case of conversion must be separately coded
- Doesn't always work like you think it will!

Polymorphism in Monomorphic Languages

- Most languages that are monomorphic have some polymorphic features:
    - Overloading
    - Coercion
    - Subtyping (range restriction in Pascal, Ada, Modula-2)
    - Value sharing (the null pointer in C)

Universal Polymorphism

- Functions work on an infinite number of types all having a common structure or property

- Goals:
  - For the writer of the function: Code things once
  - For the caller: Know it always works

- Two kinds:
  - Inclusion Subtyping and inheritance (common in OO languages)
  - Parametric Common structure

Inclusion Polymorphism

- Occurs in languages that allow subtypes and inheritance.
    - An instance of a subtype can be manipulated by the same functions that operate on instances of the supertype.
    - May or may not have specialized meaning, but always legal.
- An object belongs to many different classes that need not be disjoint (e.g., multiple inheritance).
- Examples
    - Range subtypes in Pascal, Ada, or Modula-2:

      type year = 1..7 (things that work with integers can still work with “year”s)
    - Type hierarchies in Julia, Haskell, etc.
    - Class inheritance in object-oriented languages

Parametric Polymorphism

- Originated in functional programming languages, but now common in lots of languages

- Uniformity is achieved by use of *type parameters* or *variables*

- Executes the same code for arguments of any admissible type
  (or at least appears to)

Example: Length

- Separate monomorphic functions:
  - “numLength” for list of numbers
  - “symLength” for list of symbols
  - ...
- Ugly, lots of code repetition

Example: Length

- Idea: “length” overloaded for number lists, symbol lists, ...
- Problems:
    - Doesn’t really avoid code duplication
    - Ad hoc approach
        - What about lists of ____?
    - Requires type checking we don’t have yet

Example: Length

- Idea: It’s so common, just build it in and have “length” work for any list
- What about the next thing the user wants? The next? The next after that?
- Doesn’t really help the user write things

Example: Length

- Idea: Let’s code it once for list of ____ , then let the user fill in the blank to “generate” individual functions
- Solves the code duplication problem
- But require the user to do it explicitly
- Called *explicit polymorphism* (a subclass of parametric polymorphism)

Example: Length

- Approach 1:
  - Write a function that takes a type as a parameter, then returns a function that operates on lists of that type
  - Requires that types be types—that has its own set of issues
  - Creates things at runtime

Example: Length

- Approach 2:
  - Write a template from which you can elaborate individual functions at compile time
  - Also known as *generic programming* or *templated programming*
  - Examples: Ada, C++, Java (but not at first), C# …

Example: Length

- Notation:

$$length : \forall \alpha . \mathrm{list}(\alpha) \;\; \rightarrow \;\; \mathrm{number}$$

Implicit Polymorphism

- Often found in languages that use type inference
- What if after going through all available information, we still don’t know the type signature of something?
  - If overdetermined (contradictions), that’s an error
  - If underdetermined, the program simply may not care what the type is *and will work with any type for that value*
- Example (Haskell):

```
len [ ] = 0
len (x:xs) = 1 + len xs
```

- Called *implicit polymorphism* (another case of parametric polymorphism)