# Polymorphism

CS431, Spring 2022

nprenet@bsu.edu

## Contents



Some languages (e.g. Pascal, 1970[1]) are **monomorphic**, in the sense that every value and variable can be interpreted to be of one and only one type. This can be contrasted with **polymorphic** languages, in which some values and variables may have more than one type.

- polymorphic functions are functions whose operands (actual parameters) can have more than one type
- polymorphic types are types whose operations are applicable to values of more than one type

$$\textbf{polymorphism} = \begin{cases} \textbf{universal} \begin{cases} \textbf{parametric} \\ \textbf{inclusion} \end{cases} \\ \textbf{ad} - \textbf{hoc} \begin{cases} \textbf{overloading} \\ \textbf{coercion} \end{cases} \end{cases} \tag{1}$$

## Universal vs. ad-hoc

| Universal polymorphism | Ad-hoc polymorphism |
|---|---|
| functions work on a infinite number of types, that have a common structure | functions only work on a finite set of different and potentially unrelated types |
| *same code* executes for arguments of any admissible type | function may execute *different code* for each type of argument |

[1] Pascal was the language of choice for introductory programming courses in most CS departments throughout the world during the 80s.

# How to obtain ad-hoc polymorphism

## Overloading

With **overloading**; the same variable name is used to denote different functions, and the context is used to decide which function is denoted by a particular instance of the name. C++, Java, and C# allows for overloaded **methods**, that have the same name but different signature:

```
public static class Console {
    public void WriteLine();
    public void WriteLine(string value);
    public void WriteLine(bool value);
    ...
}
```

The three functions above have the same name in the source code, but are compiled into three distinct functions. Overloading saves us the trouble of having to write and use closely related functions with different names:

```
public static class Console {
    public void WriteLine();
    public void WriteLine_s(string value);
    public void WriteLine_b(bool value);
    ...
}
```

Because **operators** are just a convenient syntactic form for functions, you can overload them as well. E.g. This piece of a C# program specifies a new meaning for the `+`, `-`, and `*` operators.

```
public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator) { ... // create new Fraction object }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);
}
```

Overloading looks like polymorphism but

- allows us to put under the same name definitions that are only remotely related
- is mostly a convenient syntactic abbreviation for a bunch of monomorphic functions.

### Coercion

With **coercion**, a semantic operation converts an argument to the type expected by a function, thus preventing a type error. They can be provided:

- statically, when the compiler adds them between an argument and a function call at compile time
- at runtime, if the type passed as an argument is determined dynamically

Coercion when the compiler automatically converts a value into a value of another type. In this piece of Java code, because `x` is declared as a `double`, integer `2` is automatically converted to a `double` value before the assignment is made.

```
double x;
x = 2;
```

The coercion could have been made explicit with a cast:

```
double x = (double) 2;
```

Coercion can be used to make functions polymorphic, through automatic conversion of their argument values to the type specified in the function declaration. E.g. Given the function signature below:

```
void f(double x){ ... }
```

All of the following calls are valid:

```
f( (byte) 1);
f( (short) 1);
f( 'a' );
f( 3 );
f( 5.6F );
```

Coercion provides some level of polymorphism but

- it is implicit, and obeys rules that typically represents a full chapter of a language's specification.
- combined with overloading, it may make the code ambiguous, as illustrated by this C bit:

  ```
  int square(int x){ return x * x ; }

  int square(double x){ return x * x ; }
  ```

  Which function will be called for `square('a')`? In this case, the *C* compiler chooses `square(int x)` (because the character type is closer to `int` than to `double`), but it would raise an error in the following case:

  ```
  int f(int x, char y){ ... }
  int f(char x, int y){ ... }
  ...
  f ('a', 'a');
  ```

## How to obtain universal polymorphism

### Inclusion polymorphism

Also called **subtyping polymorphism**, inclusion polymorphism allows for an object to belong to many different classes which need not to be disjoint (i.e. there may be inclusion of classes). If a type `Lion` is a subtype of a more general type such as `Cat`, then every object of type `Lion` can be sued in a `Cat` context in the sense that every lion **is a** cat and can be operated on by all operations that are applicable to cats. No coercion is necessary to when using an object of type `Cat` in place of an object of type `Lion`.

Altough this definition of inclusion seems obvious at first sight, its application to functions is sometimes counter-intuitive.

### How does subtyping apply to functions?

The inclusion relationship between the types of arguments handled by a function does not apply automatically to the function itself. Distinct two cases:

- functions that return polymorphic types
- functions that takes polymorphic types as parameters

### Function returning polymorphic types (covariance)

Consider the following subtyping relationships:

$$\begin{aligned} \texttt{BigCat} &\rightarrow \texttt{Cat} \\ \texttt{DomesticCat} &\rightarrow \texttt{Cat} \end{aligned} \tag{2}$$

and the following functions, with their **return** types, Haskell-style:

```
Iterate :: Cat
Iterate :: BigCat
```

In any context where `Iterate::Cat` is expected, you can safely replace with `Iterate::BigCat` or `Iterate::DomesticCat`, because the value obtained from either one will support the same subset of operations, i.e. the operations provided by the supertype `Cat`. Both the `Iterate::DomesticCat` and `Iterate::BigCat` functions are subtypes of a more general function `Iterate::Cat`:

$$\frac{\texttt{DomesticCat} \rightarrow \texttt{Cat}}{\texttt{Iterate::DomesticCat} \rightarrow \texttt{Iterate::Cat}} \tag{3}$$

The is-a relationship for the return types matches the is-a relationship for the function types: **covariance**.

### Function accepting polymorphic types (contravariance)

Consider the following functions, with their types, Haskell-style:

```
GoodPet :: Cat -> Bool
GoodPet :: DomesticCat -> Bool
GoodPet :: BigCat -> Bool
```

In a context where `GoodPet::Cat -> Bool` is expected, can you safely substitute `GoodPet::DomesticCat -> Bool`? No, you cannot. While `GoodPet::Cat -> Bool` should accept any cat, the function `GoodPet::DomesticCat -> Bool` is much pickier: in a context where any `Cat` object may be passed, passing a `BigCat` value to the substitute function will raise an error! Put otherwise: neither `GoodPet::DomesticCat -> Bool` nor `GoodPet::BigCat -> Bool` are subtypes of `GoodPet::Cat -> Bool`.

However, in a context where `GoodPet::BigCat -> Bool` is expected, can you safely substitute `GoodPet::Cat -> Bool`. Because the substitute is less choosy about its input types, it is safe to use it in a context where a narrower set of values is accepted. Let us rephrase it: `GoodPet::Cat -> Bool` is a subtype of `GoodPet::BigCat -> Bool`.

$$\frac{\texttt{BigCat} \rightarrow \texttt{Cat}}{\texttt{GoodPet::Cat -> Bool} \rightarrow \texttt{GoodPet::BigCat -> Bool}} \quad (4)$$

The is-a relationship for the function types reverses the is-a relationship of the input types: **contravariance**.

### Parametric polymorphism

A function (or a type constructor) has an implicit or explicit type parameter, which determines the type of the arguments for each application of that function. Functions that exhibit parametric polymorphism are said to be *generic*.

E.g. Consider the following Haskell function

```
length :: [a] -> Integer
```

When provided a type parameter `a`, it computes the length of a list independently from the type of its elements, doing the same kind of work for all lists; `length :: [Integer] -> Integer`, `length :: [[Char]]` are only two functions out of a set that is virtually infinite. Provided a new, custom type `Cat`, the function `length :: [Cat] -> Integer` is expected to work without code modification.

In Haskell, the type parameter is typically passed implicitly by the compiler's type inference system, such that

```
length( [1,2,3] )
```

has inferred type `length :: [Integer]`. In Java, or C#, an generic function would look like this:

```
// function declaration
Pair<S,T> combine<S,T> (S a, T b){ ... }

// function call
Pair<Integer, String>) pair = combine<Integer,String>( 2, "Hello");
```

Parametric polymorphism is the purest form of polymorphism: the same object or function can be used uniformly in different type contexts without changes, coercions, or any kind of runtime tests or special encoding of representations. However, this uniformity of behaviour requires that all data be represented, or dealt with, uniformly: by pointers (references), that is.

ML (and its direct descendant: Haskell) has been entirely built on parametric typing: a generic sorting function may for instance work on any type that supports an ordering relation.

## Conclusion: polymorphism in the wild

The four ways of relaxing monomorphism described so far are almost never observed in their pure form. Instead, almost every language implementation associates them, which makes for a complicated picture.

- the distinction between overloaded operators and functions, and coercion is typically blurry, and sometimes confusing. In C and Java, the symbol `+` may denote the integer sum, the real sum, and the string concatenation: three operations that share only an approximate similarity of algebraic structure. If you add user-defined operators, applied to user-defined types, the set of possibilities becomes infinite.
- most mainstream languages (Java, C++, C#) have constructs combine overloading, coercion, subtyping and type parameters.