Run-Time Polymorphism

CS 211

Winter 2020
polymorphism, n. (from poly- + -morphism)

1. The ability to assume different forms or shapes.
2. (biology) The coexistence, in the same locality, of two or more distinct forms independent of sex, not connected by intermediate gradations, …
3. (object-oriented programming) The feature pertaining to the dynamic treatment of data elements based on their type, allowing for an instance of a method to have several definitions.
4. (mathematics, type theory) The property of certain typed formal systems of allowing for the use of type variables and binders/quantifiers over those type variables; …
5. (crystallography) …
6. (genetics) …
let mystery xs0 =
    let rec loop acc xs =
        match xs with
        | [] -> acc
        | x :: xs' -> loop (x :: acc) xs'
    in loop [] xs0
ML stands for meta-language

```
let mystery xs0 =
  let rec loop acc xs =
    match xs with
    | []             -> acc
    | x :: xs'       -> loop (x :: acc) xs'
  in loop [] xs0
```
Ad-hoc polymorphism

```c
bool test(int v, int lo, int hi)
{
    return lo <= v && v < hi;
}

bool test(double v, double lo, double hi)
{
    return low <= v && v <= hi;
}
```
Generic = parametric + ad-hoc

template <class T>
void filter(std::vector<T>& v, T lo, T hi)
{
    size_t dst = 0;

    for (T& x : v)
        if (test(x, lo, hi))
            v[dst++] = x;

    v.resize(v.size() - dst);
}
trait Testable {
    fn test(&self, lo: &Self, hi: &Self) -> bool;
}

impl Testable for f64 {
    fn test(&self, lo: &f64, hi: &f64) -> bool {
        lo <= self && self <= hi
    }
}

fn filter<T: Testable>(
    v: &mut Vec<T>, lo: &T, hi: &T) {
    let mut dst = 0;
    for i in 0 .. v.len() {
        if v[i].test(lo, hi) {
            v.swap(dst, i);
            dst += 1;
        }
    }
}
Message/method polymorphism

Number subclass: Complex [ | realpart imagpart |

"constructor and setter omitted..."

real [ ^realpart ]
imag [ ^imagpart ]

+ other [ ^Complex real: (realpart + other real) imag: (imagpart + other imag) ]

"etc..." ]
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau \text{ is-a } \sigma$) iff every value of type $\tau$ is also a value of type $\sigma$. 

Possible examples:

- int is-a double?
- Rectangle is-a Shape?
- Square is-a Rectangle?
- vector<Rectangle> is-a vector<Shape>?
- bool(*)(Shape) is-a bool(*)(Rectangle)?
Subtype polymorphism in theory

A type $\tau$ is a **subtype** of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the *Liskov Substitution Principle*. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- `int` is-a `double`?
Subtype polymorphism in theory

A type $\tau$ is a *subtype* of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the *Liskov Substitution Principle*. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- **Integer** is-a **Real**
Subtype polymorphism in theory

A type $\tau$ is a \textit{subtype} of a type $\sigma$ (notation: $\tau$ \textbf{is-a} $\sigma$) \textbf{iff} every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the \textit{Liskov Substitution Principle}. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

\begin{itemize}
  \item Integer \textbf{is-a} Real
  \item Rectangle \textbf{is-a} Shape?
\end{itemize}
A type \( \tau \) is a subtype of a type \( \sigma \) (notation: \( \tau \textbf{ is-a } \sigma \)) iff every value of type \( \tau \) is also a value of type \( \sigma \).

(This is known as the \textit{Liskov Substitution Principle}. Restated: A function that accepts an object of type \( \sigma \) must work on objects of type \( \tau \).)

Possible examples:

- \textbf{Integer} \textbf{ is-a } \textbf{Real}
- \textbf{Rectangle} \textbf{ is-a } \textbf{Shape}
- \textbf{Square} \textbf{ is-a } \textbf{Rectangle}?
A type \( \tau \) is a *subtype* of a type \( \sigma \) (notation: \( \tau \texttt{ is-a } \sigma \)) iff every value of type \( \tau \) is also a value of type \( \sigma \).

(This is known as the *Liskov Substitution Principle*. Restated: A function that accepts an object of type \( \sigma \) must work on objects of type \( \tau \).)

Possible examples:

- Integer \texttt{ is-a } Real
- Rectangle \texttt{ is-a } Shape
- Square \texttt{ is-a } Rectangle
Subtype polymorphism in theory

A type \( \tau \) is a subtype of a type \( \sigma \) (notation: \( \tau \text{ is-a } \sigma \)) iff every value of type \( \tau \) is also a value of type \( \sigma \).

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type \( \sigma \) must work on objects of type \( \tau \).)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
- vector<Rectangle> is-a vector<Shape> ?
A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
- vector<Rectangle> is-a vector<Shape>
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau \textbf{is-a} \sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer $\textbf{is-a}$ Real
- Rectangle $\textbf{is-a}$ Shape
- Square $\textbf{is-a}$ Rectangle
- $\text{vector}<\text{Rectangle}> \textbf{is-a} \text{vector}<\text{Shape}>$
- $\text{bool} \ (\ast)(\text{Shape}) \textbf{is-a} \text{bool} \ (\ast)(\text{Rectangle})$
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
- vector<Rectangle> is-a vector<Shape>
- bool (*)(Rectangle) is-a bool (*)(Shape)
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau \textbf{is-a } \sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the \textit{Liskov Substitution Principle}. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer\textbf{is-a} Real\textbf{is-a}
- Rectangle\textbf{is-a} Shape\textbf{is-a}
- Square \textbf{is-a} Rectangle
- \textit{vector<Rectangle> is-a vector<Shape>}
- \textbf{bool} (*)(Rectangle) \textbf{is-a} bool (*)(Shape)
Subtype polymorphism in C++

```cpp
struct Base {
};

struct Derived : Base {
};
```

Then:

`Derived*` is-a `Basic*`, `Derived&` is-a `Base&`, and likewise for `const` versions, but `Derived` is-a `Base` – why not?
Subtype polymorphism in C++

```cpp
struct Base {
};

struct Derived : Base {
};

Then:

- Derived* is-a Basic*,
- Derived& is-a Base&, and
- and likewise for const versions, but
- Derived is-a Base – why not?
```
Adding “methods”

```c
struct Base {
    int f() { return 0; }
};

struct Derived : Base {
    int f() { return 1; }
};
```
Adding “methods”

```cpp
struct Base {
    int f() { return 0; }
};

struct Derived : Base {
    int f() { return 1; }
};

TEST_CASE("direct") {
    Base b;
    Derived d;
    CHECK( b.f() == 0 );
    CHECK( d.f() == 1 );
}
```
Adding “methods”

```cpp
struct Base
{ int f() { return 0; } };

struct Derived : Base
{ int f() { return 1; } };

int g(Base& b) { return b.f(); }

TEST_CASE("via reference")
{
    Base b;
    Derived d;
    CHECK( g(b) == 0 );
    CHECK( g(d) == 0 ); // ???
}
```
To determine which function to call:

- Static dispatch uses the static type of the variable
- Dynamic dispatch uses the run-time class of the object
Static versus dynamic dispatch

To determine which function to call:

- Static dispatch uses the static type of the variable
- Dynamic dispatch uses the run-time class of the object

To get dynamic dispatch in C++, a function must be `virtual`
Introducing virtual functions

```
struct Base
{ virtual int f() { return 0; } };

struct Derived : Base
{ int f() override { return 1; } };
```
Introducing virtual functions

```cpp
struct Base
{ virtual int f() { return 0; } };

struct Derived : Base
{ int f() override { return 1; } };

int g(Base& b) { return b.f(); }

TEST_CASE("via reference")
{
    Base b;
    Derived d;
    CHECK( g(b) == 0 );
    CHECK( g(d) == 1 );
}
```
— To CLion! —