Lec 15: Polymorphism

CS220: Programming Principles

Sang Kil Cha

Recap: Subtype Polymorphism

Subtype polymorphism is achieved by defining a virtual function in a base class and then overriding it in a derived class.

Polymorphism

Polymorphism

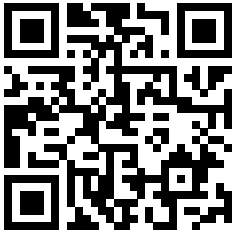
Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

- From Wikipedia

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.



Other Kinds of Polymorphism?

Can we create a function, which takes in parameters of two different types that do not have any subtype relationship?

Indeed, we have seen such functions!

Method Overloading[1]

```
type Arithmetic () =
  member __.Add (n1: int, n2: int) = n1 + n2
  member __.Add (s1: string, s2: string) = $"{s1}{s2}"

let a = Arithmetic ()
a.Add (10, 20) |> printfn "%d"
a.Add ("a", "b") |> printfn "%s"
```

[1] Should not be confused with method overriding.

Ad-hoc Polymorphism

Overloaded functions can be applied to arguments of different types, but behave differently depending on the type of the argument to which they are applied. This is called ***ad-hoc polymorphism***.

Method Overloading with Static Members

We don't really want to instantiate the `MyOp` object to invoke Add functions. To avoid that, we can use static member functions. Static members can be ***statically*** accessed ***without instantiating*** the object.

```
type MyOp () =
  static member Add (n1: int, n2: int) = n1 + n2
  static member Add (s1: string, s2: string) = $"{s1}{s2}"

MyOp.Add (10, 20) |> printfn "%d"
MyOp.Add ("a", "b") |> printfn "%s"
```

Static members look similar to functions in a module.

Operator Overloading

Operator overloading is extremely useful when dealing with user-defined types.

```
type Point (x: float, y: float) =
  member __.X = x
  member __.Y = y
  static member (+) (lhs: Point, rhs: Point) =
    Point (lhs.X + rhs.X, lhs.Y + rhs.Y)

let a = Point (1.0, 2.0)
let b = Point (3.0, 4.0)
let c = a + b // Simple and easy to understand
```

Polymorphism in Functional Programming

Polymorphic functions are everywhere in functional programming languages. This is often referred to as ***parametric polymorphism***.

```
let id x = x

val id: 'a -> 'a
```

Polymorphism in Functional Programming

Polymorphic functions are everywhere in functional programming languages. This is often referred to as ***parametric polymorphism***.

```
let id x = x

val id: 'a -> 'a
```

Polymorphism is ***not*** specific to OOP.

Polymorphism with Type Constructors

Parametric polymorphism applies to types as well. We have learned how to create a generic type with type constructor. For example, List, Set, and Map data types can take any types to construct a new type. We can then define functions that work with these generic types.

Limitation of Parametric Polymorphism

- A polymorphic function can only perform operations that are valid for all types.
- A polymorphic function cannot access the contents of the parameter.

But this is not always true if we consider dynamic typing. F# provides a way to achieve this.

Boxing

In OOP's perspective, every expression is derived from an Object (`obj`) type.

```
box 1
"hello" :> System.Object
```
is the same as `box "hello"`

```
[ box "hello"; box 1 ]
```

This is how dynamically typed languages (e.g., Python) view the world.

Unboxing

Unboxing is the process of converting a boxed value to its original type. Thus, we should know the original type of the boxed value.

```
let lst = [ box "hello"; box 1 ]
List.head lst |> unbox<string>
```

With boxing and unboxing (and with the help of reflection[2]), we can emulate dynamic typing in F#. This is why dynamically typed languages are inherently ***slower*** than statically typed languages.

[2] https://learn.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection

Duck Typing

Duck typing is a way of achieving polymorphism in dynamically typed languages. With duck typing, we only care about the methods and properties of the object, not the type of the object.

Duck Typing in Python

```
class Duck:
    def quack(self):
        print("Quack, quack")

class Person:
    def quack(self):
        print("I'm quacking like a duck")

for obj in [Duck(), Person()]:
    obj.quack()
```

Duck Typing in F#

```
type Duck () =
  member _.Quack () = printfn "Quack"

type Person () =
  member _.Quack () = printfn "I'm quacking like a duck"

[ box (Duck ()); box (Person ()) ]
|> List.iter (function
  | :? Duck as duck -> duck.Quack ()
  | :? Person as person -> person.Quack ()
  | _ -> failwith "Bad type")
```

N.B. ‘:?’ is a type test pattern. It checks if the object is of the specified type.

Can We Do Better?

We could achieve duck typing in F# with boxing and unboxing, but can we do it in a more elegant (and efficient) way?

Type Constraints

We can impose constraints on generic types in F# with `when` keyword. For example, when we look at the type signature of = operator, we see the equality constraint as follows:

```
val (=):  ('a -> 'a -> bool) when 'a :  equality
```

This means that `'a` should support equality operations.

Example Type Constraints

```
// 'T should be a subtype of System.Exception.
type MyClass<'T when 'T :> System.Exception> = class end

// 'T should have a method named Method that takes an int
// and returns an int.
type MyClass<'T when 'T: (member Method: int -> int)> =
  class end
```

There are many more. See `https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/generics/constraints`

Duck Typing Revisited

```
// Polymorphic function with a member type constraint.
let inline quack (obj: 'T when 'T: (member Quack: unit -> unit)) =
  obj.Quack ()

Duck () |> quack
Person () |> quack
```

The `inline` keyword is necessary here because F# is statically typed and the compiler cannot decide the type of `obj` at compile time. However, with inlining, the compiler can ***statically*** infer the type of the parameter from the call site and emit the appropriate code.

In-Class Activity #15

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   - `git clone https://github.com/KAIST-CS220/CS220-Main.git`
2. Move in to the directory `CS220-Main/Activities`
   - `cd CS220-Main`
   - `cd Activities`

Problem

Modify the `length` function to take an object that has a `Length` property, and return the value of the property. The function should be a polymorphic function.

Automatic Generalization

Type Inference and Automatic Generalization

F# infers the types of expressions. When it infers a function type, it determines whether a given parameter can be generic.

Automatic Generalization

```
let max a b = if a > b then a else b

val max: a: 'a -> b: 'a -> 'a when 'a: comparison
```

Limitation

Automatic generalization is performed only on (1) complete function definitions that have explicit arguments, and (2) simple immutable values. Other declarations cannot have generic types. This is often referred to as ***value restriction***.

Value Restriction Example

```
type T<'a> = { mutable V: 'a }
let x = { V = 1 }    // compile
let y = { V = None } // does not compile
```

F# compiler infers the type of `y` as `T<'a option>`, which is a generic type. However, `y` is not a function nor a simple immutable value. Thus, the compiler emits an error. If the compiler accepts `y` as a generic type, the following code will type-check:

```
y.V <- Some 42 // in File A
y.V <- Some "hello" // in File B
```

Exceptional Cases

F# compiler is smart enough to handle some exceptional cases.

```
let x = ref None // this is inferred as int option ref
x.Value <- Some 42
```

Workarounds

Typically we can work around the value restriction by (1) annotating the type explicitly, or (2) making a generic function.

```
let x = ref None // error
let x: int option ref = ref None // works

id >> id // error
let f = id >> id // error
let g x = (id >> id) x // works
```

Organizing Objects

Subtype vs. Subclass

In OOP, a subclass (child class) is a type that is derived from another type, and the subclass is considered to be a subtype of its parent class.

However, being a subclass syntax-wise does not necessarily mean that the subclass ia a ***true*** subtype of its parent, and this can lead to confusion.

True Subtype

When B is a true subtype of A, we can use B wherever A is expected.

For example, Integer is a true subtype of Number, and if we use an Integer value wherever a Number is expected in a program, the program will well behave.

Is Square a Subtype of Rectangle?

- A square is a special kind of rectangle.
- A square has all the properties of a rectangle.

Can Square replace Rectangle in all contexts without breaking the program?

Example

```
type Rectangle () =
  let mutable width = 0
  let mutable height = 0
  member _.SetWidthAndHeight (w, h) =
    width <- w
    height <- h

type Square () =
  inherit Rectangle ()

let square = Square ()
square.SetWidthAndHeight (4, 2) // This should not be allowed
```

Square is Not a True Subtype of Rectangle

A square cannot replace a rectangle in all contexts! How about method overriding?

```
type Rectangle () =
  let mutable width = 0
  let mutable height = 0
  abstract SetWidthAndHeight: int * int -> unit
  default _.SetWidthAndHeight (w, h) =
    width <- w
    height <- h

type Square () =
  inherit Rectangle ()
  override _.SetWidthAndHeight (w, h) =
    if w <> h then failwith "Bad size"
    base.SetWidthAndHeight (w, h)
```

Throwing an Exception is Bad

- It confuses the client because `Rectangle`'s `SetWidthAndHeight` method is not supposed to throw an exception.
- Throwing an exception is generally a bad practice.

Potential Solution?

Make `Square` a separate type. That is, use inheritance only when there Is a true subtype relationship. This is often referred to as ***Liskov Substitution Principle***.

Conclusion

1. Polymorphism is a powerful concept that allows us to write more general and reusable code.
2. Polymorphism is not specific to OOP.
3. Value restriction is a limitation of F# type inference, but practically it is not a big issue as we can easily work around it.

Further Readings

- https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/flexible-types
- https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/generics/statically-resolved-type-parameters

Question?