Programming Lanaugages (2)
Essence of Object-Oriented Programming

Kenjiro Taura

Classes and objects

- a *class* ≈ a data type definition + functions (*methods*) for it
- an *object* is a data instance created from a class definition

```
# define a class named rect
class rect:
  def __init__(self, x, y, width, height):
    self.x = x
    self.y = y
    self.width = width
    self.height = height

r = rect(10,20,30,40) # create an instance (or an object) of rect
```

Methods

- ≈ functions
- unlike ordinary functions, a method of the same name can be defined for multiple classes (i.e., implemented differently)

```
class rect:
  ...
  # define a method named area
  def area(self):
    return self.width * self.height

class ellipse:
  ...
  # define another method named area
  def area(self):
    return self.rx * self.ry * math.pi

```

Dynamic dispatch

- when you call a method, which method gets called among many implementations is determined by the class argument(s) belong to

```
# shapes may have both rect and ellipse instances
for s in shapes:
    ... s.area() ...
```

Language design points

```
# shapes may have both rect and ellipse instances
for s in shapes:
    ... s.area() ...
```

- in a code like the above, a variable `s` may take a value of different classes (types) over time *(polymorphism)*
- for languages that require type declarations, *how to declare/specify the type of* `s` *or* `shapes`*?*
- *does Go/Julia/OCaml/Rust require type declarations?*

Language design points

```
# shapes may have both rect and ellipse instances
for s in shapes:
    ... s.area() ...
```

- more fundamentally, how can we guarantee, prior to execution, that *type errors (≈ application of non-existing methods) do not happen at runtime?*
- such property is called *type safety*
- an algorithm that checks type safety prior to execution is often called *static type checking*
- *does Go/Julia/OCaml/Rust guarantee type safety?*

Different approaches I

1. forgo static type checking and thus type safety (e.g., Python, javascript, Lisp, Smalltalk, ...)

```
shapes = [rect(...), ellipse(...), ...]
for s in shapes:
    ... s.area() ...
```

2. disallow polymorphism altogether and make it (trivially) type-safe (e.g., Pascal)

```
rects : array of rect = [ rect(...), rect(...) ]
for s : rect in rects:
    ... s.area() ...
```

Different approaches II

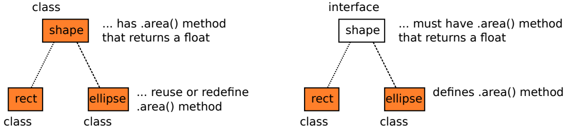
3. do some (loose) static type checking without guaranteeing type safety; allow polymorphism via unsafe casts between pointers (e.g., C)

```
void * shapes[] = { (void *)rect(...), (void *)ellipse(...) };
for s in shapes:
    ... area(s) ...
```

4. allow polymorphism yet guarantee type safety via *subtypes*
   - *C is a subtype of P ($C \leq P$)* $\equiv$ a value of C can be safely used wherever P is expected
   - allow $P \leftarrow C$ (put a value of type C in a variable of type P)

Different approaches to subtyping

- *class* vs. *interface*
  - subtype relations hold between two *classes*
  - subtype relations hold between an *interface* (or *trait*, *abstract class*, etc.) and a *class* that *implements or conforms to* it; or between two *interfaces*



- *nominal (explicit)* vs. *structural* subtyping
  - nominal : subtype relation exists only when so declared or a class is explicitly derived from the other
  - structural : subtype relation exists whenever safe (based on the structure)

How/if they guarantee type safety?

- following slides briefly explain how Go/Rust/OCaml guarantee *type safety*
- *type safety* $\equiv$ "no such methods" error never happens at runtime $\equiv$ when a program containing $o.m(\ldots)$ passes static type check, o always has method m at runtime
- recall that this is not the case for some languages (including Python, Julia, C++, etc.)

A common framework

- a type checker, *before execution*, computes (or assumes given by the programmer) the *static type* of each expression/variable
- for any *assignment-like operations* $o = p$, it gets *static types* of o $(= S)$ and p $(= T)$
- the assignment is valid $\iff T \leq S$

Note: assignment-like operations

- $\approx$ any operation in which the same value changes its static type
  - assignment to a variable/structure/array element
  - function calls (passing values to parameters)
  - function return (returning a value)

Subtype relationship

- T is a subtype of S ($T \leq S$)
- $\approx$ any value of T can be safely put anywhere S is expected
- $\approx$
  1. T has all methods S has
  2. for each method, the input type of the T's version is a *supertype* of S's
  3. for each method, the return type of the T's version is a *subtype* of S's
- note: P is a *supertype* of $Q \iff Q \leq P$ (i.e., Q is a subtype of P)

Specifically, ...

- imagine the type checker checks expression:

  $$s.m(p)$$

  where
  - s's static type is S
  - $S.m$'s input static type is P
  - $S.m$'s return static type is A
- and imagine s is assigned a value t ($s = t$) elsewhere, whose static type is T
- then
  - T must have m (obvious)
  - $T.m$'s input static type must be *supertype* of P
  - $T.m$'s return static type must be *subtype* of A

Go

- details on Assignability section of Go reference
- Q: when is a type T a subtype of another type S ($T \leq S$)?

Go

- details on Assignability section of Go reference
- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is an interface and T is a `struct/interface` that *implements* S or a pointer to it

Go

- details on Assignability section of Go reference
- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is an interface and T is a `struct/interface` that *implements* S or a pointer to it
- Q: so when is T said to *implement* an interface S?

Go

- details on Assignability section of Go reference
- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is an interface and T is a `struct/interface` that *implements* S or a pointer to it
- Q: so when is T said to *implement* an interface S?
- A:
  - T has all the methods specified in S, and
  - each method in T has the same type as the method of the same name in S

Rust

- Q: when is a type T a subtype of another type S ($T \leq S$)?

Rust

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is a reference to a *trait* and T is a reference to a `struct` that *implements* S

Rust

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
  1. S and T are identical type
  2. S is a reference to a *trait* and T is a reference to a `struct` that *implements* S
- Q: so when does T *implement* a trait S?

Rust

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following
    1. S and T are identical type
    2. S is a reference to a *trait* and T is a reference to a `struct` that *implements* S
- Q: so when does T *implement* a trait S?
- A:
    - T has all the methods specified in S, and
    - each method in T has the same type as the method of the same name in S

OCaml

- Q: when is a type T a subtype of another type S ($T \leq S$)?

OCaml

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following, among others
  1. S and T are identical type

OCaml

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following, among others
  1. S and T are identical type
  2. when each of S and T is a function type ($S = a \to b$ and $T = a' \to b'$), then $b' \leq b$ and $a \leq a'$

OCaml

- Q: when is a type T a subtype of another type S ($T \leq S$)?
- A: one of the following, among others
  1. S and T are identical type
  2. when each of S and T is a function type ($S = a \rightarrow b$ and $T = a' \rightarrow b'$), then $b' \leq b$ and $a \leq a'$
  3. when each of S and T is an object type ($S = {<}m_0 : t_0, \ldots{>}$, $T = {<}m'_0 : t'_0, \ldots{>}$), then
     - $\{m_0, \ldots\} \subset \{m'_0, \ldots\}$ and
     - for each $m_i = m'_j$, $t'_j \leq t_i$