IA010: Principles of Programming Languages

Objects

Achim Blumensath
blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Object-Oriented Programming

Original Definition

divide the program state into small parts that communicate via messages

Object-Oriented Programming

Original Definition

divide the program state into small parts that communicate via messages

Modern Definition

no consensus, but usually includes features like

- dynamic dispatch
- subtyping
- encapsulated state
- inheritance
- open recursion

Dynamic dispatch

Methods

```
next   : unit -> object
get    : unit -> int
iter   : (int -> unit) -> unit
length : unit -> int
```

Dynamic dispatch

Methods

```
next   : unit -> object
get    : unit -> int
iter   : (int -> unit) -> unit
length : unit -> int
```

Record

```
[ next   : unit -> object,
  get    : unit -> int,
  iter   : (int -> unit) -> unit,
  length : unit -> int ];
```

```
let new_empty() {
  let n =
    [ next   = fun ()  { n },
      get    = fun ()  { error },
      iter   = fun (f) { () },
      length = fun()   { 0 } ];
  n
};
let new_node(val, next) {
  [ next   = fun ()  { next },
    get    = fun ()  { val },
    iter   = fun (f) { f(val); next.iter(f) },
    length = fun ()  { 1 + next.length() } ]
};

let n1 = new_empty();
let n2 = new_node(1,n1);
let n3 = new_node(2,n2);
n3.iter(fun (x) { print "value is " x });
```

Syntactic sugar

```
object { m1 : t1 ; ... ; mk : tk ; }
⟹ [ m1 : t1 , ... , mk : tk ]

object {
  m1 ( a1 ) { b1 } ;
  ...
  mk ( ak ) { bk } ;
}
⟹
[ m1 = fun ( a1 ) { b1 },
  ...
  mk = fun ( ak ) { bk } ]
```

```
type olist =
  object {
    next   : unit -> olist;
    get    : unit -> int;
    iter   : (int -> unit) -> unit;
    length : unit -> int
  };

let new_empty() {
  let n =
    object {
      next()   { n };
      get()    { error };
      iter(f)  { () };
      length() { 0 };
    };
  n
};
```

```
let new_node(val, next) {
  object {
    next()   { next };
    get()    { val };
    iter(f)  { f(val); next.iter(f) };
    length() { 1 + next.length() };
  }
};

let n1 = new_empty();
let n2 = new_node(1,n1);
let n3 = new_node(2,n2);
n3.iter(fun (x) {
          print "value is " x });
```

Multi-methods

How to implement

```
intersect : shape * shape -> shape ?
```

Multi-methods

How to implement

```
intersect : shape * shape -> shape ?
```

First try

```
object {
  ...
  intersect(other: shape) : shape { ... } // ???
  ...
}
```

Multi-methods

How to implement

```
intersect : shape * shape -> shape ?
```

First try

```
object {
  ...
  intersect(other: shape) : shape { ... } // ???
  ...
}
```

Multi-methods

```
let intersect(x : circle,    y : circle)    : shape { ... }
let intersect(x : circle,    y : rectangle) : shape { ... }
let intersect(x : rectangle, y : circle)    : shape { ... }
let intersect(x : rectangle, y : rectangle) : shape { ... }
```

Comparison with variant types

```
type shape =
  | Point(int,int)
  | Circle(int,int,int)
  | Rectangle(int,int,int,int)
  | Group(list(shape));

let draw(sh) {
  case sh
  | Point(x,y)          => draw_point(x,y)
  | Circle(x,y,r)       => draw_circle(x,y,r)
  | Rectangle(x,y,w,h)  => draw_rectangle(x,y,w,h)
  | Group(shapes)       => iter(draw, shapes)
};
let move(sh, dx, dy) {
  case sh
  | Point(x,y)          => Point(x + dx, y + dy)
  | Circle(x,y,r)       => Circle(x + dx, y + dy, r)
  | Rectangle(x,y,w,h)  => Rectangle(x + dy, y + dy, w, h)
  | Group(shapes)       => Group(map(fun (s) { move(s, dx, dy) },
                                     shapes))
};
```

Type classes

```
typeclass Shape(a) {
  draw       : a -> unit;
  move       : a -> int -> int -> a;
  dimensions : a -> [ min_x : int, min_y : int, max_x : int, max_y : int ];
};

type point = Point(int,int);

instance Shape(point) {
  draw(Point(x,y))          { draw_point(x,y)       };
  move(Point(x,y), dx, dy) { Point(x + dx, y + dy) };
  dimensions(Point(x,y))   { [ min_x = x, min_y = y, max_x = x, max_y = y ]
};

type circle = Circle(int,int,int);

instance Shape(circle) {
  draw(Circle(x,y,r))          { draw_circle(x,y,r)        };
  move(Circle(x,y,r), dx, dy) { Circle(x + dx, y + dy, r) };
  dimensions(Circle(x,y,r))   { [ min_x = x-r, min_y = y-r,
                                  max_x = x+r, max_y = y+r ] };
};
```

Comparison

Objects

- allows inhomogeneous data structures
- each class has exactly one superclass
- easy to add new classes, tedious to add new methods
- does not support multi-methods

Type classes

- distinction between types can classes
- each type can belong to several type classes
- does support multi-methods

Variant types

- easy to add new methods, tedious to add new variants
- does support multi-methods

Subtyping

s is a **subtype** of t ($s \leq: t$) if values of type s can be used everywhere a value of type t is expected.

Subtyping

s is a **subtype** of t ($s \leq: t$) if values of type s can be used everywhere a value of type t is expected.

- **structural** (OCaml) or **by name** (Java, C++)

Subtyping

s is a **subtype** of *t* (*s* ≤: *t*) if values of type *s* can be used everywhere a value of type *t* is expected.

- **structural** (OCaml) or **by name** (Java, C++)

Basic types

`int16` ≤: `int32` ≤: `int64` or `uint16` ≤: `int32`

`uint32` ≤: `int32` or `int32` ≤: `float32` ?

Subtyping

s is a **subtype** of *t* (*s* ≤: *t*) if values of type *s* can be used everywhere a value of type *t* is expected.

- **structural** (OCaml) or **by name** (Java, C++)

Basic types

`int16 ≤: int32 ≤: int64` or `uint16 ≤: int32`

`uint32 ≤: int32` or `int32 ≤: float32` ?

Records

```
type shape     = [ x : int, y : int ];
type circle    = [ x : int, y : int, r : int ];
type rectangle = [ x : int, y : int, w : int, h : int ];
```

`circle <: shape` and `rectangle <: shape`

Functions

$f : a \to b$ and $g : (c \to d) \to e$

$g(f)$ defined?

Functions

$f : a \to b$ and $g : (c \to d) \to e$

$g(f)$ defined?

$$a \to b <: c \to d \quad \Longleftrightarrow \quad a >: c \text{ and } b <: d$$

Functions

$f : a \to b$ and $g : (c \to d) \to e$

$g(f)$ defined?

$$a \to b \leq: c \to d \quad \Longleftrightarrow \quad a \geq: c \text{ and } b \leq: d$$

- **contravariant** in a and **covariant** in b

Functions

$f : a \to b$ and $g : (c \to d) \to e$

$g(f)$ defined?

$$a \to b \leq: c \to d \quad \Longleftrightarrow \quad a \geq: c \text{ and } b \leq: d$$

- **contravariant** in a and **covariant** in b

Example

```
type shape     = [ x : int, y : int ];
type circle    = [ x : int, y : int, r : int ];
type rectangle = [ x : int, y : int, w : int, h : int ];

shape -> circle <: rectangle -> circle <: rectangle -> shape
```

Invariance

```
type box(a) = [ data : a ];

let get(box : box(a)) : a {
  box.data
};

let set(box : box(a), x : a) : unit {
  box.data := x
};
```

When is `box(a) <: box(b)`?

Invariance

```
type box(a) = [ data : a ];

let get(box : box(a)) : a {
  box.data
};

let set(box : box(a), x : a) : unit {
  box.data := x
};
```

When is `box(a) <: box(b)`?

Suppose that `box(a) <: box(b)`.

```
x : a
u : box(a)
set(u,x);          // set : box(a) -> a -> unit
let y = get(u);    // get : box(b) -> b
y : b
```

Invariance

```
type box(a) = [ data : a ];

let get(box : box(a)) : a {
  box.data
};

let set(box : box(a), x : a) : unit {
  box.data := x
};
```

When is `box(a) <: box(b)`?

Suppose that `box(a) <: box(b)`.

```
y : b
u : box(a)
set(u,y);          // set : box(b) -> b -> unit
let x = get(u);    // get : box(a) -> a
x : a
```

Encapsulated state

```
type account = object {
  deposit  : int -> unit;
  withdraw : int -> unit;
};

let new_account(balance) {
  object {
    deposit(amount)  { balance := balance + amount };
    withdraw(amount) { balance := balance - amount };
  }
};
```

```
type shape = object {
  draw        : unit -> unit;
  move        : int -> int -> unit;
  dimensions  : unit -> [ min_x : int, min_y : int,
                          max_x : int, max_y : int ];
};

let new_point(x : int, y : int) : shape {
  object {
    draw()       { draw_point(x,y)           };
    move(dx, dy) { x := x + dx; y := y + dy; };
    dimensions() { [ min_x = x, min_y = y, max_x = x, max_y = y ] };
  }
};
let new_circle(x : int, y : int, r : int) : shape {
  object {
    draw()       { draw_circle(x,y,r)        };
    move(dx, dy) { x := x + dx; y := y + dy; };
    dimensions() { [ min_x = x - r, min_y = y - r,
                     max_x = x + r, max_y = y + r ] };
  }
};
```

Inheritance

- mechanism for code reuse
- less typing
- increases maintainability (no duplicated changes)
- reduces code locality

Inheritance

- mechanism for code reuse
- less typing
- increases maintainability (no duplicated changes)
- reduces code locality

Variants/Degrees

- delegates
- adding methods
- replacing methods
- modifying methods

Delegates

```
type coloured_shape = object {
  draw       : unit -> unit;
  move       : int -> int -> shape;
  dimensions : unit -> [ min_x : int, min_y : int,
                         max_x : int, max_y : int ];
  colour     : colour;
  set_colour : colour -> unit;
};

let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
  let p = new_point(x,y);
  object {
    draw()          { set_colour(col); p.draw() };
    move()          { p.move() };
    dimensions()    { p.dimensions() };
    colour()        { c };
    set_colour(col) { c := col };
  };
```

Adding methods

```
let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
  let p = new_point(x,y);
  [ draw       = p.draw,
    move       = p.move,
    dimensions = p.dimensions,
    colour     = fun () { c },
    set_colour = fun (col) { c := col } ];

let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
  let p = new_point(x,y);
  object {
    include p;
    colour()       { c },
    set_colour(col) { c := col };
  }
```

Replacing methods

```
let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
  let p = new_point(x,y);
  [ draw       = fun () { set_colour(c); draw_point(x,y); },
    move       = p.move,
    dimensions = p.dimensions,
    colour     = fun () { c },
    set_colour = fun (col) { c := col } ];
```

Replacing methods

```
let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
  let p = new_point(x,y);
  [ draw       = fun () { set_colour(c); draw_point(x,y); },
    move       = p.move,
    dimensions = p.dimensions,
    colour     = fun () { c },
    set_colour = fun (col) { c := col } ];
```

- If another method calls `draw`, which version does it use?

Open recursion

```
type widget = object {
  draw   : unit -> unit;
  resize : int -> int -> unit;
  ...
};
let new_widget(width,height) {
  object {
    draw() { () }
    resize(w,h) { width := w; height := h; draw(); }
  }
};

type text_field = object { ... };
let new_text_field() {
  object {
    draw() { ... };
    ...
  };
};
```

Open recursion

```
type widget = object {
  draw   : unit -> unit;
  resize : int -> int -> unit;
  ...
};
let new_widget(width,height) {
  object {
    draw(obj) { () }
    resize(obj,w,h) { width := w; height := h; obj.draw(obj); }
  }
};

type text_field = object { ... };
let new_text_field() {
  object {
    draw(obj) { ... };
    ...
  };
};
```

Modifying methods

```
let new_coloured_point(x : int, y : int, c : colour) : coloured_shape =
  let super = new_point(x,y);
  [ draw       = fun () { set_colour(c); super.draw(); },
    move       = super.move,
    dimensions = super.dimensions,
    colour     = fun () { c },
    set_colour = fun (col) { c := col } ];
```

Type classes

Adding methods

```
typeclass Eq(a) {
  equal     : a -> a -> bool;
  not_equal : a -> a -> bool;
};
typeclass Eq(a) => Ord(a) {
  type cmp = | LT | EQ | GT;
  compare : a -> a -> cmp;
};

instance Eq(int) {
  equal(x,y)     { prim_equal_int(x,y) };
  not_equal(x,y) { not(equal(x,y)) };
};
instance Ord(int) {
  compare(x,y) { if x < y then LT else if x > y then GT else EQ };
};
```

Type classes

Default implementation

```
typeclass Eq(a) {
  equal     : a -> a -> bool;
  not_equal : a -> a -> bool;
  not_equal(x,y) { not(equal(x,y)) };
};

instance Eq(int) {
  equal(x,y) { prim_equal_int(x,y) };
};
```

Multiple inheritance

- allows code reuse from several classes
- makes objects more complicated:
  - are common base classes shared or distinct?
  - which super class is used for a method call?

Object types

```
type shape = object type {
    draw : unit -> unit;
    move : int * int -> unit;
    dimensions : unit -> rectangle;
  };

let new_point(x : int, y : int) : shape {
  object {
    draw()       { ... };
    move(dx, dy) { ... };
    dimensions() { ... };
  }
};
```

Mixins

function $F : I \to J$ mapping classes $A : I$ to $F(A) : J$

```
type coloured_shape = object {
  draw       : unit -> unit,
  move       : int -> int -> shape,
  dimensions : unit -> [ min_x : int, min_y : int,
                         max_x : int, max_y : int ],
  colour     : colour,
  set_colour : colour -> unit
};

let make_coloured(s : shape, c : colour) : coloured_shape =
  [ draw       = fun () { set_colour(c); s.draw() },
    move       = s.move,
    dimensions = s.dimensions,
    colour     = fun () { c },
    set_colour = fun (col) { c := col } ];
```

- can replace (multiple) inheritance

Discussion

- common problem: single mechanism combining all features
  - makes system complicated
  - causes conceptual confusion (e.g., 'is-a' vs. 'has-a')
- better: separate mechanisms
  - cleaner code
  - only use needed features
- object-oriented programming is powerful but complex
- simpler solutions (e.g., higher-order functions) are often better
- boiler-plate code gives false sense of productivity
- object-oriented design leads to low performance
  (poor cache locality)