

## Programming Languages and Compilers (CS 421)

---

Elsa L Gunter  
2112 SC, UIUC  
<http://www.cs.uiuc.edu/class/sp07/cs421/>

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

## How to encode Objects with Functions?

---

- Functional Languages have fairly straightforward semantics
- Object Oriented Languages are more common
- Problem: How to encode in functional language?
  - To understand their semantics
  - To be able to simulate objects in a language without them

Elsa L. Gunter

## What is an Object?

---

- Data (state) and functions (interface) are grouped together.
- Functions have their own local state
- Objects can send and receive messages
- Objects can refer to themselves
- Object Oriented Programming is a programming language paradigm that facilitates defining, handling and coordinating objects.

Elsa L. Gunter

## Preliminaries

---

- We will use the following functions:  
**let pi1 (x,y) = x**  
**let pi2 (x,y) = y**  
**let report (x,y) = print\_string "Point: ";**  
    **print\_int x;**  
    **print\_string ",";**  
    **print\_int y;**  
    **print\_newline ();**  
**let movept (x,y) (dx,dy) = (x+dx,y+dy)**

Elsa L. Gunter

## Point with State

---

```
let mktPoint myloc =  
let myloc = ref myloc in  
( myloc,  
  (fun () -> pi1 !myloc),  
  (fun () -> pi2 !myloc),  
  (fun () -> report !myloc),  
  (fun dl -> myloc := movept !myloc dl)  
)
```

Elsa L. Gunter

## Point with State

---

- mktPoint creates a point with local state
- Defines a tuple of functions that share a common state.

```
let (lref,getx,gety,show,move) =  
  mktPoint (2,4);;
```

- It is cumbersome to use.

Elsa L. Gunter

## Working with the Point object

```
let (lref,getx,gety,show,move) = mkPoint (2,4);;
```

```
# getx ();;  
- : int = 2  
# move (3,4);;  
- : unit = ()  
# show ();;  
Point: 5,8  
- : unit = ()
```

Elsa L. Gunter

## Improvement - Use Records

```
type point =  
{  
  loc : (int * int) ref; getx : unit -> int;  
  gety : unit -> int; draw : unit -> unit;  
  move : int * int -> unit;  
}
```

Elsa L. Gunter

## Improvement - Use Records

```
let mkPoint newloc =  
let myloc = ref newloc in  
{ loc = myloc;  
  getx = (fun () -> pi1 !myloc);  
  gety = (fun () -> pi2 !myloc);  
  draw = (fun () -> report !myloc);  
  move =  
  (fun dl -> myloc := movept !myloc dl)  
}
```

Elsa L. Gunter

## Working with the Point object

How do you instantiate the object ?

```
let point = mkPoint (2,4);;
```

How do you invoke the function getx?

```
# point.getx ();;  
- : int = 2
```

How do you invoke the function move?

```
# point.move (2,3);;  
- : unit = ()
```

Elsa L. Gunter

## Adding self

```
let mkPoint newloc =  
let rec this =  
{ loc = ref newloc;  
  getx = (fun() -> pi1 ! (this.loc));  
  gety = (fun() -> pi2 ! (this.loc));  
  draw = (fun() -> report ! (this.loc));  
  move = (fun dl ->  
    this.loc := movept ! (this.loc) dl) }  
in this;;
```

Elsa L. Gunter

## Memory

- The record point references to the fields
- If you copy a point, the data does not get copied!

```
# let p1 = mkPoint (4,7);;  
val p1 : point = {loc={contents=4, 7}; ...}  
# let p2 = mkPoint(6,2);;  
val p2 : point = {loc={contents=6, 2}; ...}
```

Elsa L. Gunter

## Memory

```
# let p3 = p1;;
val p3 : point = {loc={contents=4, 7}; ...}
# p1.move(5,5);;
- : unit = ()
# p3;;
- : point = {loc={contents=9, 12}; ...}
```

Elsa L. Gunter

## So far...

- We used a record to implement a type for points.
  - Advantages:
    - Every method had its own name and type.
    - Simple syntax for manipulating the object.
    - It's fast: we know at compile time which method is been called.
  - Disadvantages
    - Inheritance is very difficult with this model.
      - Adding a new message type means updating everything.

Elsa L. Gunter

## Message Dispatching

- Object is kind of data that can receive messages from program or other objects.
  - Need implementation where type doesn't change when new methods are added.
- Let a point object be a function that takes a string and returns an appropriate matching for that string.

Elsa L. Gunter

## mkPoint

```
let mkPoint x y =
  let x = ref x in let y = ref y in
  fun st -> match st with
  | "getx" -> (fun _ -> !x)
  | "gety" -> (fun _ -> !y)
  | "movx" -> (fun nx -> x := !x + nx; !x)
  | "movy" -> (fun ny -> y := !y + ny; !y)
  | _ -> raise(Failure("Unknown message."))
```

- All methods now have to have type `int -> int`

Elsa L. Gunter

## Using mkPoint

How do you instantiate the object ?  
let point = mkPoint (2,4);;

How do you invoke the function getx?

```
# point "getx" 0;;
- : int = 2
```

How do you invoke the function move?

```
# point "movx" 2;;
- : int = 4
```

Elsa L. Gunter

## Adding a new method

- Exercise: How would we add a `report` method?

```
# let mkPoint x y = ...
  fun st -> match st with
  .....
  | "report" -> (fun _ -> print_string "X = ";
                  print_int !x;
                  print_string "\n";
                  print_string "Y = ";
                  print_int !y;
                  print_string "\n";0)
  | _ -> raise (Failure("Function not understood"));
```

Elsa L. Gunter

## Adding `this`

- Exercise: How would we add `this`?

```
# let mkPoint x y = let this = ...
  (fun st -> match st with
    .....
    | _ -> raise (Failure("Function not understood")))
  in this;;
```

Elsa L. Gunter

## Example: `fastpoint` subclass

Three entities involved: the superclass (`superpoint`) and the subclasses (`point`) and (`fastpoint`). `fastpoint` moves twice as fast as the original point

What does it mean for `fastpoint` to be a subclass of `superpoint`?

- `fastpoint` should respond to the same messages.
  - It may override some of them.
  - It may add its own.
  - It may **not** remove any methods.

Elsa L. Gunter

## Implementing

- Point construction needs to return the “*public*” data to `fastpoint` and `point`.
- `fastpoint` returns a dispatcher:
  - If `fastpoint` dispatcher can handle a message, it does.
  - Otherwise, it sends *the message* to `point`.

Elsa L. Gunter

## Code: `superpoint`

```
let mkSuperPoint x y =
  let x = ref x in let y = ref y in
  ((x,y),
  fun st -> match st with
    "getx" -> (fun _ -> !x)
  | "gety" -> (fun _ -> !y)
  | "movx" -> (fun nx -> x := !x + nx; !x)
  | "movy" -> (fun ny -> y := !y + ny; !y)
  | "report" -> (fun _ -> report (!x, !y); 0)
  | _ -> raise (Failure ("Function not
understood")));;
```

Our instance variables are now public.

Elsa L. Gunter

## Code: `point`

```
let mkPoint x y =
  mkSuperPoint x y;;
```

Elsa L. Gunter

## Code: `fastpoint`

```
let mkFastPoint x y =
  let ((x,y),super) = (mkSuperPoint x y) in
  ((x,y),
  fun st -> match st with
    "movx" -> (fun nx -> x := !x + 2 * nx; !x)
  | "movy" -> (fun ny -> y := !y + 2 * ny; !y)
  | _ -> super st);;
```

Elsa L. Gunter

## Code: `fastpoint`

- This technique is flexible
  - We can add methods very easily.
- But it's also slow
  - Imagine if we had a chain of 20 classes...

Elsa L. Gunter

## Till now...

- Have implemented objects using message dispatch model.
- More Limitations :-
  - Had to make a member public in order to be accessed in sub-class.
  - No notion of “protected” member.

Elsa L. Gunter

## Polymorphism

- Polymorphism: same function name used at different types
- Adhoc Polymorphism
  - Different operations on different types using the same name.
  - e.g.: - sum (int x, int y), sum (float x, float y)
  - Different function for each instance

Elsa L. Gunter

## Structural Polymorphism

- One algorithm, one compiled code unit used at different types it, based on outermost structure of argument
- Type of polymorphism in OCaml

Elsa L. Gunter

## Inheritance polymorphism

```
# let p1,p2, p3, p4 =((mkPoint 2 3), (mkPoint 3 2),  
                  (mkFastPoint 5 3), (mkFastPoint 3 9));;
```

```
# List.map (fun pt -> pt "report" 0)  
          [p1, p2, p3, p4];;
```

Point: 2,3  
Point: 3,2  
Point: 5,3  
Point: 3,9

point

fastpoint

The function passed to map will use both point and fastpoint types

Elsa L. Gunter

## Discussion: Dynamic Dispatch

- Java uses “every object is of type Object” technique.
- Strong type system makes it cumbersome to simulate objects -- have to either
  - define a new type to encompass all objects, or
  - force all methods to have same type
- Can't handle dynamic dispatch (aka dynamic binding).
  - Need to have each method take the object as an argument

Elsa L. Gunter

## Discussion: Class variables

---

- Have only discussed instance variables
- Class variables are variables shared by all instances of class.
- Only one copy of class variables:-
  - Can implement class variables in OCAML, using global variables.

Elsa L. Gunter

## Conclusions

---

- Objects have a lot of flexibility, and allow us to create useful abstractions.
- They can be implemented using functions.
- These are useful enough in practice, and difficult enough to implement that most modern languages now include them, including OCAML. ('O'-CAML)
- An alternative to to Objects is a flexible module system
  - Main ingredient missing: dynamic binding

Elsa L. Gunter