CMSC 330: Organization of Programming Languages

Objects and Abstract Data Types

Abstract Data Types

- Expose signature
  - operators to create, combine, and observe values
- Hide representation & implementations

- Advantages
  - Good engineering (libraries, code reuse, local reasoning)
  - Optimizations (choose implementation w.r.t. your code)

- Limitations
  - Require static typing
  - Different representations cannot mix

Objects

- Object-oriented programming (OOP)
  - Computation as interactions between objects

- An object...
  - Is a collection of fields (data)
  - ...and methods (code)
  - When a method is invoked
    - Method has implicit this parameter that can be used to access fields of object

Relating ADTs to Objects

```
type set
val empty    : set

val isEmpty  : set -> bool
val insert   : set -> int -> set
val contains : set -> int -> bool
```

Abstract Data Types: define behavior for all sets

Observations on set argument

```
type iSet = {
  isEmpty     : bool;
  insert      : int -> iSet;
  contains    : int -> bool;
}
```

“Objects” here:
- encoded as records
- fields define how this object interacts with other objects

ADT for Int Sets using Closures

```
module IntSetF : INT_SET =
struct
  type set = bool * (int -> bool)
    (* whether set is empty,
       returns true if given int in set *)

  let empty = (true, fun x -> false)
  let isEmpty (e,_) = e
  let insert (_,f) i =
    (false, if f i then f
            else fun j -> i = j || f j)
  let contains (_,f) i = f i
end;;
```

Objects by Encoding

- We will consider an encoding of objects similar to the Object → FP encoding we saw before
  - Objects contain methods
    - ➢ But we will define “top level” methods that take the object as an argument. These will call the object’s methods
  - This will let us see how to encode the self/this parameter
    - ➢ Not something we looked at before
- In our examples, objects are *not* imperative
  - “Updates” to an object just return a new object with the change made from the original

IntSet as an object

```
type iSet =  {
  isEmpty      : bool;
  insert       : int -> iSet;
  contains     : int -> bool;
}
```

```
# let (x:iSet) = … ;;
- …
# let y = x.insert 1;;
- …
# let z = insert_obj y 2;;
- … (* calls y's insert method *)
# y.contains 1;;
- : bool = true
# z.contains 2;;
- : bool = false
```

Example: Insert Set

calling contains field of the object s

```
let rec insert_obj s n =
 if s.contains n then s else {
  isEmpty  = false;
  contains = fun i -> (i = n || s.contains i);
  insert   = fun i -> insert_obj this i;
}
```

recursive call to the object being defined

Question: How can we encode `this`?

Answer: Use the fixpoint combinator Y

```
fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

Example: Insert Set

calling contains field of the object s

```
let rec insert_obj s n =
 if s.contains n then s else (fix (fun this _ -> {
  isEmpty  = false;
  contains = fun i -> (i = n || s.contains i);
  insert   = fun i -> insert_obj (this ()) i;
})) ()
```

recursive call to the object being defined

Question: How can we encode `this`?

Answer: Use the fixpoint combinator Y

`fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`

Quiz 1: Which is the Empty Set ?

```
A. let empty_obj _ = (fix (fun this _ ->  {
     isEmpty = false;
     contains = fun i -> false;
     insert = fun i -> insert_obj (this()) i })) ()
```

```
B. let empty_obj _ = (fix (fun this _ ->  {
     isEmpty = true;
     contains = fun i -> false;
     insert = fun i -> insert_obj (this()) i})) ()
```

```
C. let empty_obj _ = (fix (fun this _ ->  {
     isEmpty = true;
     contains = fun i -> false;
     insert = fun i -> this() })) ()
```

Quiz 1: Which is the Empty Set ?

```
B. let empty_obj _ = (fix (fun this _ ->  {
    isEmpty = true;
    contains = fun i -> false;
    insert = fun i -> insert_obj (this()) i})) ()
```

Quiz 2: Which produces the union?

A.
```
let union_obj s1 s2 = (fix (fun this _ ->  {
    isEmpty  = s1.isEmpty || s2.isEmpty;
    contains = fun i -> s1.contains i || s2.contains i;
    insert   = fun i -> insert_obj (this()) i })) ()
```

B.
```
let union_obj s1 s2 = (fix (fun this _ ->  {
    isEmpty  = s1.isEmpty || s2.isEmpty;
    contains = fun i -> s1.contains i && s2.contains i;
    insert   = fun i -> insert_obj (this()) i})) ()
```

C.
```
let union_obj s1 s2 = (fix (fun this _ ->  {
    isEmpty  = s1.isEmpty && s2.isEmpty;
    contains = fun i -> s1.contains i || s2.contains i;
    insert   = fun i -> insert_obj (this()) i})) ()
```

The Union Set: Autognosis

```
C. let union_obj s1 s2 = (fix (fun this _ ->  {
     isEmpty  = s1.isEmpty && s2.isEmpty;
     contains = fun i -> s1.contains i || s2.contains i;
     insert   = fun i -> insert_obj (this()) i})) ()
```

Autognosis: An object can only access other objects through their public interface

Problem: The representation of s1 and s2 is unknown, so … cannot used for optimizations

Solution: Expose more implementation details

Lack of Autognosis in ADTs

Autognosis: An object can only access other objects through their public interface

- \+ Each implementation can inspect representation
  - \+ Used for optimizations (eg., knowing the representation is a sorted lists)
- \- Different implementations cannot interact
  - \- eg., cannot union **`IntSet`** and **`IntSetBST`**

Quiz 3: The Even Set

```
A. even_obj _ = (fix (fun this _ ->  {
     isEmpty = false;
     contains = fun i -> i mod 2 = 0;
     insert   = fun i -> insert_obj (this()) i })) ()
```

```
B. even_obj _ = (fix (fun this _ ->  {
     isEmpty = true;
     contains = fun i -> true;
     insert   = fun i -> insert_obj (this()) i })) ()
```

```
C. even_obj _ = (fix (fun this _ ->  {
     isEmpty = i mod 2 = 0;
     contains = fun i -> i mod 2 = 0;
     insert   = fun i -> insert_obj (this()) i })) ()
```

The Even Set: Flexibility

```
A. even_obj _ = (fix (fun this _ ->  {
     isEmpty = false;
     contains = fun i -> i mod 2 = 0;
     insert   = fun i -> insert_obj (this()) i })) ()
```

Flexibility:

- Objects accept any value that implements required methods
- They can be easily extended with new (here, *infinite*) representations.

Computations are Object Interactions

```
(union_obj
  (even_obj())
  (empty_obj().insert(3))
).contains(3)
```

Computation via *dynamic binding*:

- The function to be called is selected from the object record
- Every object has a different **`contains`** field.
- Difficult to reason about which will be called

Computations are Object Interactions

contains = fun i -> i mod 2 = 0

```
(union_obj
   (even_obj())
   (empty_obj().insert(3))
).contains(3)
```

contains = fun i -> false

contains = fun i -> i == 3 || false

contains = fun i -> i == 3 || false || i mod 2 = 0

Quiz 4: ADTs vs. Objects

Used for data abstraction
(i.e., separate behavior from implementation)

A. ADTs
B. Objects
C. None
D. Both

Quiz 4: ADTs vs. Objects

Used for data abstraction
(i.e., separate behavior from implementation)

ADTs use Type Abstraction:
expose a type whose representation is hidden.

Objects use Procedural Abstraction:
expose procedures available on each object.

A. ADTs
B. Objects
C. None
D. Both

Quiz 5: ADTs vs. Objects

Require Static Type System

A. ADTs B. Objects

C. None D. Both

ADTs require Static Types

```
type set
val empty    : set

val isEmpty  : set -> bool
val insert   : set * int -> set
val contains : set * int -> bool
```

A. ADTs B. Objects
C. None D. Both

Quiz 5: ADTs vs. Objects

Require Static Type System

ADTs rely on Static Type System to define the Type Abstraction

Objects define abstractions via records:
can be used in Static & Dependently typed lang.

A. ADTs B. Objects
C. None D. Both

Quiz 6: ADTs vs. Objects

Allow mixing different representations.

A. ADTs
B. Objects
C. None
D. Both

Quiz 6: ADTs vs. Objects

Allow mixing different representations.

ADTs doesn't permit interaction of different representations. Representation can be inspected, which is used for optimizations.

Objects cannot inspect representations (example union). Different representations can be mixed.

A. ADTs
B. Objects
C. None
D. Both

ADTs vs. Objects

ADTs are easier to reason about statically

- The implementation is statically known (unlike dynamic binding)
- Obey theories in math (abstract algebra) and PL (existential typing)

Objects are flexible and easy to extend.

- Only requirement to have appropriate fields
- Different representations can interact, so precise behavior hard to reason about, statically