A Theory of Objects
(M. Abadi & L. Cardelli)

CS520: Programming Languages
Santiago Pericas-G.
Boston University
Fall 2001
Chapter 3:
Advanced Class-based Features
Object Types

- It is widely recognized that keeping *specifications* separate from *implementations* has many advantages.

- This separation can be achieved by making a distinction between classes and *object types*.

- This is the reason why we have been writing, e.g., `InstanceTypeOf(cell)` instead of `cell`. 
Object Types

• A consequence of subsumption is that a type such as InstanceTypeOf(cell) may refer to objects of a subclass of cell.

• Instances of these subclasses may have additional attributes as well as overridden methods.

• Thus, strictly speaking, instances of subclasses of cell are not really cells, but they have type InstanceTypeOf(cell).
Object Types

- InstanceTypeOf(cell) is meta-notation for:

  ObjectType Cell is
  
  var contents: Integer;
  
  method get(): Integer;
  
  method set(n: Integer);

  end;
InstanceTypeOf(reCell) is meta-notation for:

```plaintext
Object Type ReCell

method get(): Integer;
method set(n: Integer);
method restore();

data var contents: Integer;
var backup: Integer;
```

end;

Subclassing vs. Subtyping

• A *subtyping* relation can be defined independently from the *subclassing* relation.

• One possible definition:

  \[ A' <: A \text{ if } A' \text{ has the same components as } A \text{ and possibly more} \]

• It follows from this definition that:

  \[ \text{InstanceTypeOf(reCell)} <: \text{InstanceTypeOf(cell)} \]
Subclassing vs. Subtyping

• A consequence of this definition of subtyping is that:

  If \( c' \) is a subclass of \( c \) then
  \[
  \text{InstanceTypeOf}(c') <: \text{InstanceTypeOf}(c)
  \]

• What about the converse? Does not hold in general (c.f. next slide). Consequently, \textit{subtyping-implies-subtyping} instead of \textit{subclassing-is-subtyping}.
Subclassing vs. Subtyping

- Suppose we define:
  
  ```
  class myNewCell is
    var contents: Integer := 0;
    method get(): Integer is ... end;
    method set(n:Integer) is ... end;
    method clear() is ... end;
  end;
  ```

- Given this definition, we have 
  
  `InstanceTypeOf(myNewCell) <: InstanceTypeOf(cell)`
  
  even though `myNewCell` is not a subclass of `cell`. 
Consider the following object types: with the assumption that Vegetables <: Food.

**ObjectType Person is**

```
  ...
  method eat(food:Food);
end;
```

**ObjectType Vegetarian is**

```
  ...
  method eat(food:Vegetables);
end;
```
Type Parameters

- Does the relation Vegetarian <: Person hold? No, as this implies that Food <: Vegetables since food is an argument to the method eat and, therefore, in contravariant position.

- There are two solutions to this problem:
  - bounded type parameterization,
  - bounded abstract types.

- Instead of defining object types, we define operators over types (cf. next slide).
Bounded Type Parameterization

ObjectOperator PersonEating[F<:Food] is
  ...
  method eat(food:F);
end;
ObjectOperator VegetarianEating[F<:Vegetables] is
  ...
  method eat(food:F);
end;

• From these definitions, it follows that for every type parameter \( F <: \) Vegetables
  VegetarianEating[F] <: PersonEating[F]
Binary Methods

• Even with Self types, there are cases in which subtyping is not possible (i.e. unsound).

• This happens when the type Self occurs in a negative (i.e. contravariant) position.

• A method that takes an argument of type Self is known as a binary method.
Binary Methods

class maxClass is
  var n:Integer := 0;
  method max(other:Self):Self is ... end;
end;
subclass minMaxClass of maxClass is
  method min(other:Self):Self is ... end;
end;

Because Self occurs negatively, we cannot have

  InstanceTypeOf(minMaxClass) <: InstanceTypeOf(maxClass)
A number of alternative solutions have been proposed during the last few years.

One possible solution is the use of Object Protocols (c.f. Section 3.5).

The previous example shows that the are cases where subclassing (attibute inheritance) may be undesirable even without subsumption.