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

CS520: Programming Languages
Santiago Pericas-G.
Boston University
Fall 2001
Chapter 2:
Class-based Languages
Classes and Objects

- Class-based languages are centered around the notion of a **class**.

- A class describes the structure of a set of instances or **objects**.

- Objects are created using a special constructor commonly known as **new**.
Classes and Objects

class cell is
  
  var contents:Integer := 0;
  method get():Integer is
    return self.contents;
  end;
  method set(n:Integer) is
    self.contents := n;
  end;
end;
Classes and Objects

Figure 2-1. Naive storage model.
Classes and Objects

- Creating objects:

  ```
  var myCell:InstanceTypeOf(cell) := new cell
  ```

- Using objects:

  ```
  procedure double(aCell:InstanceTypeOf(cell)) is
    aCell.set(2 * aCell.get());
  end;
  ```
Method Lookup

- For space-efficiency reasons, methods are factored into *method suites*.
Method Lookup

- In the presence of inheritance, method lookup may require examining a chain of method suites.
- An occurrence of the keyword self within a method refers to the object that originally received the invocation.
- In general, methods cannot be extracted as functions (unsound) and cannot be updated (not necessarily unsound).
Subclasses and Inheritance

• An extension to the notion of a class is that of a subclass.
• In a subclass:
  • Fields can be replicated or added.
  • Methods can be replicated, added or overriden.
• Inheritance is the sharing of attributes between a class and its subclasses.

inheritance ≠ subclassing
Classes and Objects

subclass reCell of cell is
  var backup:Integer := 0;
  override set(n:Integer) is
    self.backup := self.contents;
    super.set(n);
  end;

method restore() is
  self.contents := self.backup;
end;
end;
Subclasses and Inheritance

- **Observation**: Without subclasses, the keyword `self` in a class declaration always refers to an instance of that class.

- In many languages, a special keyword `super` can be used to invoke the old version of a method.

- In the presence of subclassing, a special `method lookup mechanism` is needed.
  - This mechanism depends on whether `static type info` is available or not.
Subclasses and Inheritance

Figure 2-4. Collapsed method suites.
Subsumption

- Subclassing usually comes with the associated notion of subsumption.

Example:

```plaintext
var myCell:InstanceOfTypeOf(cell) := new cell
var myReCell:InstanceOfTypeOf(reCell) := new reCell
procedure f(x:InstanceOfTypeOf(cell)) is ... end;
myCell := myReCell;
f(myReCell);
```
Subsumption

- Type checker’s rule:

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

  which is derived as follows:

  - If \( a : A \) and \( A <: B \) then \( a : B \)
  - \( c' \) is a subclass of \( c \) \textit{iff} \( \text{InstanceTypeOf}(c') <: \text{InstanceTypeOf}(c) \).
Subsumption

- Method invocation (revisited):
  ```
  procedure g(x:InstanceTypeOf(cell)) is
    x.set(3);
  end;
  g(myReCell);
  ```

- Static vs. dynamic dispatch:
  - `x.set(3)` executes `set` in class `cell` (static)
  - `x.set(3)` executes `set` in class `reCell` (dynamic)
Type Info, Lost and Found

- Even though (in general) subsumption has **no run-time effect**, it has the effect of **reducing static knowledge** about the true type of an object.

- **Example**: When subsuming from `InstanceTypeOf(reCell)` to `InstanceTypeOf(cell)`, the ability to access the field `backup` is lost.

- Information can be recovered in two ways:
  - Via dynamic dispatch,
  - Via the use of a special construct such as `typecase` (**akin to `instanceof` in Java**).
Depth Subtyping

- Subtyping in depth must be defined for every `type constructor` in the language.
- The operands of a type constructor can be subtyped in three different ways: `covariant`, `contravariant` and `invariant`.
- Which form of depth subtyping applies to each type constructor depends on the `operations` defined over terms having that type.
Depth Subtyping

- Read-only Pairs: operations $\text{fst}(p)$ and $\text{snd}(p)$. Subtyping is \textit{covariant} on each component.

$$A \times B <: A' \times B' \iff A <: A' \text{ and } B <: B'$$

- Suppose $\text{int} <: \text{real}$ (i.e. $[[\text{int}]] \subseteq [[\text{real}]]$). Then,

$$\text{int} \times \text{int} <: \text{int} \times \text{real}$$
Depth Subtyping

- Functions: the only operation is function application. Subtyping is *contravariant* on the domain and *covariant* on the range.

\[ A \rightarrow B <: A' \rightarrow B' \text{ iff } A' <: A \text{ and } B <: B' \]

- Suppose `int <: real` (i.e. \([\text{[int]}] \subseteq [\text{[real]}]\)). Then,

\[ \text{real} \rightarrow \text{int} <: \text{int} \rightarrow \text{real} \]
Method Specialization

- In most class-based languages, method overriding is \textit{invariant}. This can be relaxed to allow for \textit{method specialization}.

```
class c is
    method m(x:A):B is ... end;
    method m1(x:A):B is ... self ... end;
end;
subclass c’ of c is
    override m(x:A’):B’ is ... end;
end;
```
Method Specialization

procedure f(o:InstanceTypeOf(c)) is
    ... o.m(a) ...
end;
f(new c');

How should A, A’ and B, B’ be related?
A can be generalized to A’ (i.e. A <: A’) and B can be specialized to B’ (i.e. B’ <: B). That is, because methods are non-updatable, they can be inherited covariantly. In our example

A’ → B’ <: A → B
Self

- Implicit method specialization by inheritance:
  - In method m1 of class c, self has type InstanceTypeOf(c).
  - In method m1 of class c’, which is inherited from class c, self has type InstanceTypeOf(c’).

- Hence, implicitly:

  *The type of self is covariantly specialized on inheritance.*
Self Types

Consider the following example:

```plaintext
class c is
  var x:Integer := 0;
  method m():InstanceTypeOf(c) is ... end;
end;
subclass c’ of c is
  var y:Integer := 0;
end;

In most cases, we want the return type of m in class c’ (inherited from c) to be InstanceTypeOf(c’). However, this is in general unsound.
```
Self Types

Suppose method m in class c is defined as follows,

\[
\text{method m(): InstanceTypeOf(c) is return } aC; \text{ end;}
\]

where aC is a (proper) instance of class c. Now, consider the following code:

\[
\text{var o: InstanceTypeOf(c') := new c';}
\]

\[
o.m().y; \quad (* \text{aC has no field y! *})
\]
Self Types

Solution: in class c define m as follows

\[
\text{method } m() : \text{Self is } \ldots ; \text{ return } E ; \text{ end;}
\]

where expression E must be derivable from self. For example,

- It can be self itself, as self has type Self
- It can be a call to another method returning type Self
- ...

It can be self itself, as self has type Self