Introduction

- Classifying the object-oriented notions
- Class-based languages
  - Subclasses and inheritance
  - Subsumption and dynamic dispatch
  - Subtyping
- Object-based languages
  - Embedding and Delegation

Class-Based Languages

- A class is intended to describe the structure of all the objects generated from the class.
- Internally an object can be understood as a reference to a record of attributes.
- An object can be created of a class c by the construction **new** c.

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

- Distinction between classes and types.
- For space efficiency method code is not directly embedded into objects.
- Instead, methods are factored into *method suites* that are shared by the objects of the same class. (delegation)

Subclasses and Inheritance

- A subclass describes also the structure of a set of objects.
- This happens incrementally, as extensions and changes to its direct superclass.
- Inheritance is the implicit sharing of attributes between a class and its subclasses.
- However distinguished from subclassing, since one may choose to override everything and thus inherit nothing.
- Inheritance affects method lookup, delegation-based languages use hierarchical method suites, in statically typed languages the suites are collapsed.

Subsumption and Dynamic Dispatch

- (subtype) polymorphism from the point view of the typechecker
- A reflexive and transitive subtype relation ($<:$) between $InstanceOf$ types satifies the properties (2) and (3).
  - (1) If c' is a subclass of c, and $o' : InstanceTypeOf(c')$, then $o' : InstanceTypeOf(c)$.
  - (2) If $a : A$ and $A <: B$, then $a : B$.
  - (3) $InstanceTypeOf(c') <: InstanceTypeOf(c)$ iff c' is a subclass of c.
- Together (2) and (3) yield (1).

- Property (2), i.e. a value of of type A can be viewed as a value of supertype B, is called *subsumption.*
- Property (3), subclassing-is-subtyping is the characteristic property of subtype relations in classical class-based languages.
- Dispatch: (re-examine the meaning of method invocation)
  Static dispatch: $x.set(3)$ executes the code of set from class $cell$.
  Dynamic dispatch: $x.set(3)$ executes the code of set from class $reCell$.
- Dynamic dispatch is found in all object-oriented languages.

Type Information, Lost and Found

- Subsumption reduces the static knowledge of the true type of an object.
- How to utilize information lost in subsumption ?
- Purist approach: Dynamic dispatch is the only mechanism.
- However, most languages provide a way of examining the true type (*typecase*) of an object.
- Typecase regarded as impure for 3 reasons.
  1. It violates the object abstraction by revealing private information.
  2. It renders programs more fragile, by introducing a form of dynamic failure.
  3. It makes code less extensible.

Co-, Contra- and Invariance

- Review of basic subtyping properties of product and function types.
- For a product type $A \times B$ with extractable components we get:

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

- We say that $\times$ is a covariant operator, since $A \times B$ varies in the same sense as A or B.
- For function types $A \to B$ (argument type A, result B we get:

$$A \to B <: A' \to B' \text{ provided that } A' <: A \text{ and } B <: B'$$

- We say that $\to$ is a contravariant operator in its left argument because $A \to B$ varies in the opposite sense as A.

- Finally, consider pairs, whose components can be updated, denoted $A \quad B$. We get:

$$A \quad B <: A' \quad B' \text{ provided that } A = A' \text{ and } B = B'$$

- Since the operator does not enjoy co- or contravariance properties, it is called *invariant* in both of its arguments.
- These inclusion rules justified by the definition of subtyping and subsumption.

Method Specialization

- Allow an overriding method to adopt different arguments and result types.

```
class c is
  method m(x:A):B is...end;
end;
subclass c' of c is
  override m(x:A'):B' is...end;
end;
```

- When o' of type $InstanceOf(c')$ is subsumed into $InstanceOf(c)$ and $o'.m(a)$ is invoked, the argument may have the type A.
- Therefore it is sufficient to require $A <: A'$ and $B' <: B$.

- That is, the parameter type is generalized and the result type specialized.
- Another form of specialization happens implicitly by inheritance.
- The type **self** is silently specialized (covariantly).
- In general, the inclusion rules for inheritance are the opposite.
- That is, parameter types may be specialized and result types generalized.

Self Type Specialization

- Class types are often recursive, in the sense that the definition of class c may contain definitions of $InstanceTypeOf(c)$.
- If e.g. a method call returns $InstanceTypeOf(c)$ an extending class c' cannot soundly specialize this to $InstanceTypeOf(c')$.
- However, by returning the type of **self**, denoted **Self** the desired effect is obtained.
- **Self** may be used in covariant positions, contravariant uses are unsound.

Object Types

- Traditional class-based languages have a strict correlation between inheritance, subclassing and subtyping.
- Introduce types for objects that are independent of classes.
- Thus, the separation of object specification and implementation can be achieved.
- Without object types, an object having $InstanceTypeOf(c)$ may contain none of the code of class c.
- With object types two classes c and c' may produce the same object type, satisfying a certain *protocol*.

Distinguishing Subclassing from Subtyping

- With object types new definition of subtyping is needed.
- Structural subtyping:

  $O' <: O$ if O' has the same components as O and possibly more

- With this definition objects support multiple subtyping, and we have:

  (4) If c' is a subclass of c, then
  $ObjectTypeOf(c') <: ObjectTypeOf(c)$.

- Subsumption still based on subtyping.
- Later subclassing is separated from subtyping, decreasing the flexibility of subsumption.

Type Parameters

- A technique used to remedy typing difficulties due to contravariance.
- Object types are converted to type operators (functions from types to types) to get sound subtyping relations.
- A related technique is called bounded abstract types.
- A parameter is added as an attribute together with the typing constraints.

Object Protocols

- Find a relation between the type induced by a class and the type induced by one of its subclasses, in the absence of subclassing-is-subtyping.
- Use an object operation T-$Protocol$ abstracting the recursive occurrences of T, e.g.

```
ObjectOperator MaxProtocol[X] is
  var n: Integer;
  method max(other:X):X;
end;
```

```
ObjectOperator MinMaxProtocol[X] is
var n: Integer;
method max(other:X):X;
method min(other:X):X;
end;
```

Let $\prec:$ denote the higher-order subtype relation between type operators:

$$P \prec: P' \text{ iff } P[T] \prec: P'[T] \text{ for all types } T$$

- For the subprotocol relation we have two possible options:

$$S \text{ subprotocol } T \text{ if } \; S <: T\text{-}Protocol[S]$$
$$\text{or}$$
$$S \text{ subprotocol } T \text{ if } \; S\text{-}Protocol \prec: T\text{-}Protocol$$

- Programming languages may define a matching relation $< \sharp$ directly over types.

- In the context above, subsumption is lost.

Object-Based Languages

- Intended to be more simple and flexible than class-based languages.
- Classes are replaced by constructs for creating individual objects.
- However, a construct like new possible:

```
procedure newCell(m: Integer):Cell is
  object cell:Cell is
    var contents:Integer := m;
    method get():Integer is return self.contents end;
    method set(n:Integer) is self.contents := n end;
  end;
  return cell;
end;

var cellInstance:Cell = newCell(0);
```

Prototypes and Clones

- In prototype-based languages stock objects are created from prototypical ones and customized later.
- The basic mechanism for this is cloning.
- One may customize both fields and methods.

```
var cellClone:Cell := clone cellInstance;
cellClone.contents := 3;
cellClone.get :=
  method():Integer is
    if self.contents < 0 then return 0 else return self.contents end;
  end;
```

Embedding and Delegation

- Inheritance may be considered as extension of existing objects.
- Not an easy task:
  - clashes of the additional attributes with existing ones
  - modifying the representation of an object while, maintaining it's identity
- Most commonly extended objects are built by building new objects that include attributes from existing ones.
- A donor is the object from which the attributes are obtained.
- A host is the object to which the attributes are incorporated.
- Obtaining may happen *implicitly* or *explicitly.*
- Incorporation may happen by *embedding* or *delegating*

Delegation

- Delegation is the redirection of field access and method invocation from an object or a prototype to another.
- A child contain a *parent link*.
- All children share the attributes of their parent.
- When a child is constructed, it is assumed that the parent is statically known.
- (Explicit) delegation provides a clean way of delegating operations to multiple objects.
- Implicitly this can be done by listing multiple parents.

- Embedding and delegation fundamentally distinct ways of achieving inheritance with prototypes.
  - In delegation methods and fields may be updated and the changes are seen in all inheriting hosts.
  - It is often permitted to replace a donor link with another one in an object.
  - Cloning is still taken to perform shallow copies.
- Delegation is the most common of the two.
- Criticism for dynamic webs of dependencies that lead to fragile systems.
- The question of the best model of inheritance still open.

Dynamic Inheritance, Mode-Switching and Traits

- Static Inheritance: parent links are fixed for all time.
- Dynamic Inheritance: parent links may be updated.
- Mode-switching: a parent is replaced with an object having the same attributes.
- Separation of object roles in classless languages:
  - Traits: parents for prototype objects, begin possibly incomplete.
  - Prototypes: intended as object generators via cloning.
  - Objects: intended to be used and carry local state.

Types for Object-Based Languages

- Dynamic features hard to capture on static type systems.
- Typed prototype-based languages strongly restrict the dynamic features.
- Mode-switching can be typed rather easily.
- The treatment of method specialization and Self types require consideration.
- Contravariant **Self** still problematic, protocol mechanisms needed.