4/11/17

CS 345 Programming Languages

13: Object Oriented Programming

Remember Abstract Data Types (ADTs)

- Combining data with associated operations

Haskell:

```haskell
module Stack (Stack, empty, isEmpty, push, top, pop) where
empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a, Stack a)
```

Remember Record-Subtyping

\[
\{ x: \{ a: \text{Int}, b: \text{Bool} \}, y: \{ m: \text{Int} \} \} <: \{ x: \{ a: \text{Int} \}, y: \{ \} \}
\]

\[
\frac{k_i \leftarrow \{ x: \{ a: \text{Int} \}, y: \{ \} \}}{\text{for each } i} \frac{\text{Gamma} \vdash t_i : T_i}{\text{Gamma} \vdash k_i : S_i \leftarrow \{ a: \text{Int}, y: \{ \} \}}
\]

\[(\text{SA-Rcd})\]

\[
\frac{\text{Gamma} \vdash t_z : T_z}{\text{Gamma} \vdash \{ l_i = t_i \ldots l_n = t_n \} : \{ l_i : T_i \ldots l_n : T_n \}}
\]

\[(\text{TA-Rcd})\]

Objects

- Class = Template from which Objects can be created
- Object = Instance of a Class.

- Encapsulation
  - Data is hidden from direct manipulation from the outside
- Subtyping
- Inheritance
  - Subclasses inherit behavior from the superclass, can selectively override where desired
- Dynamic Dispatch
  - Object determines which implementation of a polymorphic method is called
Encapsulation

- **Underlying principle: Modularity**
  - Minimize interdependencies among Modules
  - Maximizing reuse

- **Data abstraction**
  - External interface describes abstract methods

- **The “Closed” in the Open-Closed Principle**
  - Class should be open for extension but closed for modification

---

Example

- Java:

  ```java
  public class Stack <E> {
      private ArrayList<E> list = new ArrayList<E>();
      public void push(final E elem) {
          list.add(0, elem);
      }
      public E pop () {
          return list.remove(0);
      }
  }
  ```

---

Subtyping

- **We know subtyping for functions:**
  - Contravariance for arguments of subtype
  - Covariance for return types of subtype

- **These rules are syntactical**
  - Rules describe *safe substitution* based on syntax

- **Sometimes, syntactic equivalence is not enough**

---

Subtyping

```java
interface Rectangle {
    void setSize(int w, int h)
}
```

```java
interface Square extends Rectangle {
    void setSize(int w, int h) throws IllegalArgumentException
}
```

- Is Square a valid subtype of Rectangle?
Behavioral subtyping

- Behavioral subtyping is semantic:
  - The two syntactical rules
  - Contravariance of argument types
  - Covariance of return types
  - No new exceptions can be thrown by the subtype
  - Pre-conditions cannot be strengthened by the subtype
  - Post-conditions cannot be weakened by the subtype
  - Invariants must be preserved
  - History constraint: New methods introduced by the subtype may not allow state changes that were not allowed in the methods of the supertype

Liskov Substitution Principle

- If we have strong behavioral subtyping, the following (called Liskov Substitution Principle or LSP) holds:
  - Let $q(x)$ be a property provable about objects $x$ of type $T$. Then $q(y)$ should be provable for objects $y$ of type $S$ where $S$ is a subtype of $T$.

Inheritance

class A {
  int a;
  float b;
  void foo();
}

class B extends A {
  double c;
}

Inheritance

- Reusing functionality by specializing a superclass
  - Creates a hierarchy of classes which are in an “is-a” relationship
  - Derived classes can override methods of the superclasses.
Example:
```java
class A {
    int a;
    float b;
    void foo();
}
class B extends A {
    double c;
    @Override
    void foo();
}
```

Subtyping $\neq$ Inheritance
- Subtyping requires structural equivalence
  - Only then we can safely substitute
- Inheritance creates a nominal subtype which may or may not also be a structural subtype
- Many language restrict inheritance to what is permissible as structural subtypes, hence the confusion.
- Eiffel and C++ are examples of languages which allows inheritance that violates the structural subtype relationship with the superclass.

Inheritance in Scala
```scala
class Point(xc: Int, yc: Int) {
    val x: Int = xc
    val y: Int = yc
    def move(dx: Int, dy: Int): Point =
        new Point(x + dx, y + dy)
}
class ColorPoint(u: Int, v: Int, c: String) extends Point(u, v) {
    val color: String = c
    override def move(dx: Int, dy: Int): ColorPoint =
        new ColorPoint(x + dx, y + dy, color)
}
```

Dynamic Dispatch
- The Smalltalk model for method calls:
- Dynamic dispatch
  - Decision at runtime by the object
Dynamic Dispatch

Consider the following class (in Java):

```java
public abstract class StringBuffer {
    public void append(Integer i);
    public void append(Float f);
    public void append(String s);
    public void append(Object o);
}
```

```java
StringBuffer buf = new StringBuffer();
```

What happens in Java?:

```java
Integer i = new Integer(10);
buf.append(i);
String s = new String("foo");
buf.append(s);
Object o = new String("foo2");
buf.append(o);
```

From this perspective: static dispatch, compiler has already generated the right calls (see next slide)

Calls method with the right number of arguments and the most specific parameter type, based on the declared type of the argument.

---

Dynamic Dispatch

Constant pool:

```
#16 = Class              
#17 = Utf8               test/StringBuffer ...
#24 = Methodref         #16.#25
#25 = NameAndType       #26:#27
#26 = Utf8              append
#27 = Utf8              (Ljava/lang/Object;)V
```

Code:

```
aload_1
aload_2
invokevirtual #24 test/StringBuffer(Ljava/lang/Object;)V
```

Dynamic Dispatch

```java
public class Foo {
    public void callme() { }
    public void dontcallme() { }
}
```

```java
public class Bar extends Foo {
    public callme() { }
}
```

```java
Foo f = new Bar();
f.callme();
```
Single Dispatch
- Bar.callme is called.
- Invokevirtual is dispatched to the actual object and it will invoke its own method (or lowest method in the inheritance chain).
- Single Dispatch: The question which method is called for x.callme(y);
- depends only on the runtime type of x.
- Also called a virtual function call.

Multiple Dispatch
- If the call of x.call(a,b,c,..)
- depends on the runtime type of x and all the arguments a,b,c,... we have multiple dispatch
- Such methods are called Multimethods
- Natively supported by few languages, e.g., CLOS (Common Lisp Object System), Clojure, etc.

```
(defmethod callme ((x foo) (a integer))
(defmethod callme ((x bar) (a integer))
(defmethod callme ((x bar) (a number))
```

VTables
- Common implementation:
- Every class gets a virtual method table at runtime which contains method signatures and addresses to the method implementation
- vtable for Foo:

<table>
<thead>
<tr>
<th>method</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>callme()V</td>
<td>&amp;Foo.callme()V</td>
</tr>
<tr>
<td>dontcallme()V</td>
<td>&amp;Foo.dontcallme()V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>method</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>callme()V</td>
<td>&amp;Bar.callme()V</td>
</tr>
<tr>
<td>dontcallme()V</td>
<td>&amp;Foo.dontcallme()V</td>
</tr>
</tbody>
</table>