First-class Functions and Patterns

Corky Cartwright
Stephen Wong
Department of Computer Science
Rice University

Encoding First-class Functions in Java

- Java methods are *not* data values; *they cannot be used as values.*
- But java classes include methods so we can implicitly pass methods (functions) by passing class instances containing the desired method code.
- Moreover, Java includes a mechanism that closes over the free variables in the method definition!
- Hence, first-class functions (closures) are available implicitly, but the syntax is wordy.
- Example: Scheme **`map`**

Interfaces for Representing Functions

For accurate typing, we need different interfaces for different arities. With generics, we can define parameterized interfaces for each arity. In the absence of generics, we will have to define separate interfaces for each desired typing.

`map` example:

```
/** The type of unary functions in Object -> Object. */
interface Lambda {
  Object apply(Object arg);
}

abstract class ObjectList {
  ObjectList cons(Object n) {
    return new ConsObjectList(n, this);
  }
  abstract ObjectList map(Lambda f);
}
   ...
```

Representing Specific Functions

- For each function that we want to use a value, we must define a class, preferably a singleton. Since the class has no fields, all instances are effectively identical.
- Defining a class seems unduly *heavyweight*, but it works in principle.
- In OO parlance, an instance of such a class is called a strategy.
- Java provides a lightweight notation for singleton classes called anonymous classes. Moreover these classes can refer to fields and **`final`** method variables that are in scope. In DrJava language levels, all variables are **`final`**. **`final`** fields and variables cannot be rebound to new values after they are initially defined (immutable). **`final`** methods cannot be overridden.
- Anonymous class notation:

```
new <type>() {
  <member_1>
  ...
  <member_n>
}
```

Anonymous Class Example

```
new Lambda() {
  Object apply(Object arg) {
    return EmptyObjectList.ONLY.cons(arg);
  }
}
```

There are pending proposals to provide better notation for lambda abstractions. For now, you must pay attention to the interface signature defined in the library/program you are using.

This interface (`Lambda`) together with its implentation classes is called the strategy pattern. This pattern enables us to represent varying behavior. Within an object, a field of this type can be bound to any unary function.

Another Example: Building Sort Objects

- Recall the `IntList` class from Lecture 20. Assume that we want to create sort functions that sort `IntList`s in different ways (using different algorithms and orderings). How can create such objects and how can we apply them to `IntList`s? By
  - Defining a `Sorter` interface

    ```
    interface Sorter {
      IntList sort(IntList host);
    }
    ```

  - Defining a hook in `IntList` for applying `Sorter` objects to `this`.
  - Defining strategy objects that implement `Sorter`.

Note: we are introducing an interface specific to this problem (instead of using `Lambda)` to support a more precise typing. If `Lambda` had generic (parameterized) type, there would be no advantage to creating a special interface; we could use an instantion of type `Lambda` instead..

Naive Coding of `UpSorter`

```
class UpSorter implements Sorter {
  private UpSorter() { }
  IntList sort(IntList host) {
    if (host.equals(EmptyIntList.ONLY)) return host;
    ConsIntList cHost = (ConsIntList) host;
    return insert(sort(cHost.rest()), cHost.first());
  }
  IntList insert(IntList host, int elt) {
    if (host.equals(EmptyIntList.ONLY))
      return EmptyIntList.ONLY.cons(elt);
    ConsIntList cHost = (ConsIntList) host;
    if (elt <= cHost.first()) return cHost.cons(elt);
    return insert(cHost.rest(), elt).cons(cHost.first());
  }
}
```

What Is Ugly About Class `UpSorter`?

- The `Sorter` interface is essentially a copy of `Lambda`.
- `UpSorter` defines sorting code statically using a decision tree of predicates, just like a functional program. Ugh … no inheritance. Not OO.
- How can we do better? Need to introduce (*i*) λ-abstractions called *visitors* with the same internal structure as the interpreter pattern and (*ii*) hooks in our `IntList` class, namely `accept` methods, that apply visitors to `this`. Special closure objects called visitors that let us represent the interpreter pattern code for a method on `IntList` as a first-class data object. A visitor is simply a closure (first-class function representation) for a composite with a clause (method) for each variant (concrete class) in the composite.
- What is this extra work accomplish?
  * Decouples operations on composites from composites.
  * Inheritance!

Defining **upSort** Using Interpreter

```
abstract class IntList { …
  abstract IntList upSort();
  Abstract IntList insert(int i);
}
class EmptyIntList extends IntList { …
  IntList upSort() { return this; }
  IntList insert(int I) { return cons(i); }
}
class ConsIntList extends IntList { ...
  IntList upSort() { rest.upSort().insert(first); }
  IntList insert(int i) {
    if (i <= first) return cons(i);  /* this.cons(i) */
    return rest.insert(i).cons(first);
  }
}
```

Deconstructing the Interpreter Pattern

- In the interpreter pattern, the method is declared as **`abstract`** in the root class/interface and defined concretely in each concrete variant (subclass). To package the code for a method defined by the interpreter pattern in a separate closure object (a *visitor*) we need to write concrete method definitions corresponding to the interpreter pattern:

```
class ...Visitor {
  Object forEmptyIntList(EmptyIntList host) {
   ... <EmptyIntList method code for upSort using host instead of this> ...
  }
  Object forConsIntList(ConsIntList host) {
   ... <ConsIntList method code for upSort using host instead of this> ...
  }
}
```

Visitors with Arguments

Two approaches:

- Design accept to take an argument list. Disadvantages:
  - Must pass empty argument list to 0-ary visitors.
  - Cannot accurately type visitor arguments forcing extra casting.
- Embed the arguments in the visitor. Disadvantages:
  - Unnatural for programmers who do not want to think in functional terms. Functional analog: currying and partial application (with receiver as last argument.
  - New visitors (instead of new argument lists) have to be constructed for recursive calls with different arguments. No efficiency penalty since visitors with arguments have same (often smaller) footprint as argument lists!

My preference: embed the arguments. Why? I like functional thinking. More importantly, accurate typing plays a pivotal role In software engineering.

Defining upsort as a visitor cont.

```
abstract class IntList {
  abstract Object accept(IntListVisitor v);
  …
}
Class EmptyIntList {
  …
  Object accept(IntListVisitor v) { return v.forEmptyIntList(this); }
  …
}
Class ConsIntList {
  …
  Object accept(ConsIntList v) { return v.forConsIntList(this); }
  …
}
interface InListVisitor {
  Object forEmptyIntList(EmptyIntList host);
  Object forConsIntList(ConsIntList host);
}
class UpSorter implements IntListVisitor { …
  Object forEmptyIntList(EmptyIntList host) { return host; }
  Object forConsIntList(ConsIntList host) {
    IntList sortedRest = (IntList) host.rest().accept(this);
    return sortedRest.accept(new Inserter(host.first()));
  }
}
class Inserter implements IntListVisitor {
  Int i;
  Object forEmptyIntList(EmptyIntList host) { return host; }
  Object forConsIntList(ConsIntList host) {
    if (i <= host.first()) return host.cons(i);
    IntList insertedRest = (IntList) host.rest().accept(this);
    return insertedRest.cons(host.first());
  }
}
```

Invoking Visitors

The corresponding composite class must include hooks to invoke visitors for the class. To specify the signature of these hooks, we need to introduce an `accept` method in all composite classes and an interface for all visitors that operate on `IntList`

```
interface IntListVisitor {
  Object forEmptyIntList(EmptyIntList host)
  Object forConsIntList(ConsIntList host)
}
```

The hook methods have trivial definitions:

```
abstract class IntList { ...
  abstract Object accept(IntListVisitor v);
}
class EmptyIntList extends IntList { ...
  Object accept(IntListVisitor v) { return forEmptyIntList(this); }
}
class ConsIntList extends IntList { ...
  Object accept(IntListVisitor v) { return forConsIntList(this); }
}
```

The Role of Visitors

- OO formulation of closure for a composite.
- Assumes that composite Is not likely to change often.
- Remember: problem decomposition is not affected by using visitor pattern; only the syntax!
- Why use visitors? There is a compelling reason in addition to "elegance". It the same reason why the interpreter pattern is far superior to static method definitions: **inheritance**.
- What is the primary software engineering disadvantage of visitors (and OO in general)? Dynamic dispatch makes tracing code during debugging difficult.

`UpSort` Example

- Go to DrJava
- See code files saved with this lecture in the course wiki.

Reprise: Anonymous Classes

- What do free variables mean inside anonymous classes. What do they mean in λ-expressions?
- In Java, the free variables can be either:
  - fields, or
  - `final` local (method) variables.
- Use them in doing the `filter` problem in HW8.

For Next Class

- Master the strategy pattern and anonymous classes; you need them for HW8.
- Start trying to code with visitors; you will use them extensively in HW9.
- Please report any problems with the DrJava Functional Language Level.