Higher-Order Functional Programming in Java

Corky Cartwright
Department of Computer Science
Rice University

Partial Hoisting

- In a union hierarchy, the same code may be repeated in *some proper subset* of the variants.
- We can eliminate this code duplication by introducing a new abstract class that is a *superclass only of the variants that repeat the same code.*
- Partial hoisting modifies the form of the class diagram because it introduces a new abstract class below the *root* (parent) abstract class of the union.

Data Domain Definitions

- Functional programs typically manipulate *algebraic data types* (inductively defined trees) sometimes augmented by *functions as data values*. A context free grammar where the right hand sides of productions denote trees rather than strings is a good model (called a *tree grammar*). Regrettably tree grammars are not part of the standard "theory" curriculum in undergraduate computer science.
- We use the *composite pattern*, the recursive generalization of the *union pattern*, to represent algebraic data types (ignoring function values for the moment). The composite pattern is simply a very important special case of the union pattern where one of more fields in a variant (clause in the inductive definition) has the same type as the parent type, providing a mechanism for constructing arbitrarily large data values.
- Each different form of value construction in the definition is typically a separate Java class; hence a Java *class* plays roughly he same role as a Racket *struct*. The parent type is typically an interface or abstract class. (Since Java 8, the methods in interfaces can be concrete, so we presumably can always use interfaces.) The `IntList` class hierarchy from the current homework assignment is a good example.

The Interpreter Pattern

- To define a method `m` on a composite class, we follow the same process as we would in defining a method on a union class, with one new wrinkle. In the variants that refer to the composite class (have fields of composite class type), computing `m` for embedded self references will usually involve delegating the task of computing `m` to the parent composite class which uses dynamic dispatch to determine what code is executed. Dynamic dispatch corresponds to case-splitting as in the Racket **cond** construct or pattern matching in the ML-languages.
- In the `IntList` code provided in the current homework, the only embedded reference to `IntList` in variant subclasses is the `rest` field in `ConsIntList`. In the interpreter pattern we recursively apply `m` to fields of the parent class type.
- The Interpreter pattern is simply structural recursion in the context of object-oriented data (the composite pattern)

Example: IntLists

- An `IntList` is either:
  - `EmptyIntList()`, the empty list, or
  - `ConsIntList(first,rest)`, a non-empty list, where `first` is an `int` and `rest` is an `IntList`.
- Some examples include:

```
EmptyIntList()
ConsIntList(7,EmptyIntList())
ConsIntList(12,ConsIntList(17,Empty()))
```

IntList

```
abstract class IntList { }

  class EmptyIntList extends IntList { }

  class ConsIntList extends IntList {
    int first;
    IntList rest;
}
```

Defining Methods on IntList

Sort example:

```
abstract class IntList {
  abstract IntList sort() { }
}
class EmptyIntList extends IntList {
  IntList sort() { ... }
}
class ConsIntList extends IntList {
  int first;
  IntList rest;
  IntList sort() { ... }
}
```

IntList sort cont.

```
abstract class IntList {
  abstract IntList sort() { }
  abstract IntList insert(int i) { }
}
class EmptyIntList extends IntList {
  IntList sort() { return this; }
  IntList insert(int i) { return new ConsIntList(i, this); }
}
class ConsIntList extends IntList {
  int first;
  IntList rest;
  IntList sort() { return rest.insert(first); }
  IntList insert(int i) {
    if (i <= first) return new ConsIntList(i, this);
    else return new ConsIntList(first, rest.insert(i));
}
```

IntList sort cont.

```
abstract class IntList {
  abstract IntList sort() { }
  abstract IntList insert(int i) { }
}
class EmptyIntList extends IntList {
  IntList sort() { return this; }
  IntList insert(int i) { return new ConsIntList(i, this); }
}
class ConsIntList extends IntList {
  int first;
  IntList rest;
  IntList sort() { return rest.sort().insert(first); }
  IntList insert(int i) {
    if (i <= first) return new ConsIntList(i, this);
    else return new ConsIntList(first, rest.insert(i));
}
```

Three Important Idioms

Singleton Pattern

- 0-ary variants (no fields) typically have only one instance, *e.g.*, the empty list.
- Idiom for creating one and only one instance.

Strategy Pattern

- Functions as data values can be represented by instances of anonymous inner classes.
- Idiom for dynamically creating new function values.b

Parametric Polymorphism (Generic types)

- Classes (and methods) can be parameterized by type
- Regrettably type parameters are not first-class; usage is restricted relative to ordinary types

Singleton Pattern

- In Java, a **final** method variable or field cannot be modified once it is bound. Idea: bind a static final field to the sole instance of a class and make the constructor **private**.
- Example: **EmptyIntList**

```
class EmptyIntList extends IntList {
  static final EmptyIntList ONLY = new EmptyIntList();
  private EmptyIntList() { }
  IntList sort() { return this; }
  IntList insert(int n) { return cons(n); }
}
```

- To refer to the empty list, write **EmptyIntList.ONLY**.

Strategy Pattern

- In Java 1.1 (the first revised release of Java), inner classes were added to the language. We will ignore static inner classes since they only change the visibility of raw class names (without the package name qualifier); they have semantics identical to ordinary top-level classes. The interesting case is the "dynamic" inner class where every instance of such a class has an "enclosing instance", an instance of the enclosing class. in most common usages the enclosing class is the class of `this`.
- I think the inventor of Java inner classes, John Rose, a close friend of Guy Steele when he was an MIT graduate student, understood that inner classes where the OO-analog of closures (the representation of a function value in a language supporting functions as data). He even included notation for anonymous inner classes analogous to lambda-abstractions in Scheme/Racket. So an anonymous inner class is conceptually a closure like a lambda-abstraction.

Strategy Pattern cont.

- What is a closure? The code for a lambda-abstraction plus an environment specifying the values of the free variables.
- I think the inventor of Java inner classes, John Rose, a close friend of Guy Steele when he was an MIT graduate student, understood that inner classes where the OO-analog of closures (the representation of a function value in a language supporting functions as data). He even included notation for anonymous inner classes analogous to lambda-abstractions in Scheme/Racket. So an anonymous inner class is conceptually a closure like a lambda-abstraction.
- John's original proposal allowed arbitrary references to free method variables within anonymous inner classes, but this forced variables that appear free in anonymous inner classes to have heavier-weight implementations than ordinary method variables, so the Java standards committee within Sun Microsystems decided to only allow free references to **`final`** variables. Why? They can be copied as hidden fields in the anonymous inner class object because they are immutable!

Strategy Pattern cont.

- How do we represent a function as an anonymous inner class. We introduce and interface for the particular function type we need. Stephen used to support a library in Comp 310 of such interface (parameterized by generic types which we will avoid for now). Say that we want to represent a function from int to int. Then the interface

```
interface FunctionInt_Int {
  public int apply(int x);
}
```

  should suffice.

- An anonymous inner class extends a type (typically an interface) filling in any method that is not yet defined. The code for the method is simply the code to compute the desired function! The Java compiler concocts a garbled name for the anonymous inner class and only one instance is ever created (unless a programmer digs out the garbled name …)

Parametric Polymorphism

- In Racket, we quickly realized that restricted the elements of a list to a single type like number was a bad idea in most contexts because we were forced to replicate potentially shareable code for every different kind of list. So we introduced annotations that were parametric.
- Exactly the same situation arises in all statically typed languages, even Java with its more flexible type system that most statically typed languages. Initially, Java limped by because type `Object` is the supertype of all objects (instances of classes), but using it to support code sharing and flexibility was clumsier than in dynamically typed languages (like Racker) because most methods in Java require specific subtypes of `Object` implying that such polymorphic code required lots of casts as soon as specific values were extracted from polymorphic data structures. In addition, Sun claimed Java was the “last programming language” (no one really believed it) but Java needed to compete with the ML-family of languages and with C++ which had an ugly form of type parameterization called templates.
-

Parametric Polymorphism cont.

- In 1997-1998, programming language researchers proposed several different scheme for adding type parameterization (called “generic types” by the OOP research community). Guy Steele and I proposed a rather elegant scheme to support first-class generic types but it was rejected (too complex) and a scheme proposed by a group including Phil Wadler and Martin Odersky won based on type erasure.
- I doubt that either Phil or Martin is very proud of what has ensued. Java type parameters are erased from the byte code generated by the Java compiler (the byte code has no provision for supporting parametric type information) so many natural uses of parametric types are forbidden in Java. Nevertheless, there are reasonable workarounds in many cases and stinky, passable workarounds in others which enable most developers to hold their noses and get by.
- Scala managed to clean things up somewhat but even Scala is hobbled by its compatibility with Java and it ugly generic type system. Martin Odersky was interested in exploring the adaptation of the Cartwright-Steele proposal for Scala but my planned sabbatical was derailed so Scala and Java both live with crippled type systems.

Parametric Polymorphism cont.

- The javac compiler enforces generic typing, but the compiler allows that system to be breached in various ways via annotations and "raw" types. I think newer versions of Java have partially plugged some holes but Java will never be a language with a rigorous type discipline in practice. There are many situations where cheating on type-checking yields far more elegant implementations.
- Fundamental limitation: no typable code can rely on run-time type information that is not revealed by control flow and even some of this information (*e.g.*, the results of `instanceof` tests) is not available.
- Examples of operations that are forbidden where `T` is type parameter
  - `new T(…), new T[], (T) <expr>, (<generic type>) <expr>`
- To escape these restrictions, the parameterized types of objects must be discoverable at run-time, which the Cartwright-Steele proposal supported at essentially no run-time overhead cost.

Working Around Type Erasure

- Best approach: use an intuitive understanding of parameterized types (a la Racket annotation) and back off (weakening your types) when bitten by type errors reported by the compiler.
- Most important trick: use `ArrayList<T>` when you might want `T[]`.
- Wildcard types (generic types with ? used as a parametric type) are really ugly and hard to work worth; I would avoid them if possible. They are designed to accommodate subtyping at the level of generic type variables, *e.g.* `ArrayList<Integer>` is a subtype of `ArrayList<Number>`. In most cases, the extra work to get the wildcard types right is not worth the effort. It is probably better to simply live with weaker typing. (Potential exception: widely used libraries. But usage is more difficult.)