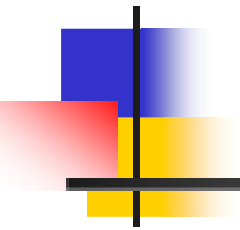
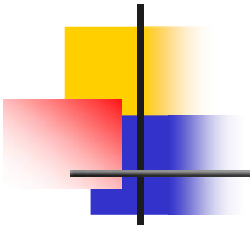


Higher-Order Functional Programming in Java

Corky Cartwright

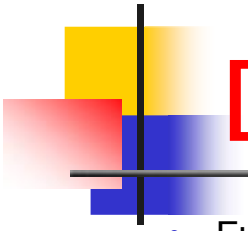
Department of Computer Science

Rice University



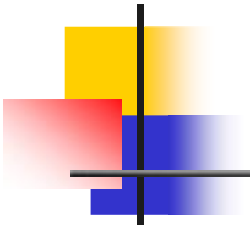
Partial Hoisting

- 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.



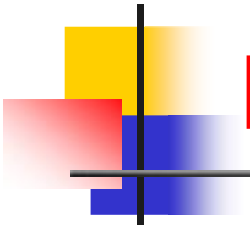
Data Domain Definitions

- Functional programs typically manipulate *algebraic data types* (inductively defined trees). A context free grammar (called a *tree grammar*) where the right hand sides of productions denote trees rather than strings is a good intuitive model if we allow function values to be leaves. Note that all of the intermediate node types correspond to constructor functions. In mathematics, the resulting algebra is called a *free term algebra* (if we exclude functions as leaves). Regrettably tree grammars are not part of the standard "theory" curriculum in undergraduate computer science. They are incredibly important in mathematics but they are so simple and straightforward that the corresponding mathematical subject (called "universal algebra") is rarely the focus of a formal course. (Most of the interesting problems in universal algebra are really problem in programming language theory!)
- We use the *composite pattern*, the recursive extension 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*, except that classes also impose an inheritance hierarchy on the methods supported by the classes in a composite. 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 at the cost of some syntactic ugliness.) The `IntList` class hierarchy from HW4 and HW5 is a good example.



The Interpreter Pattern

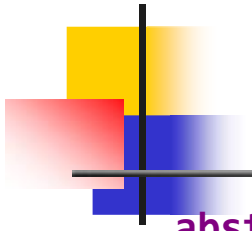
- To define a method `m` on a composite class, we follow the same process as we would in defining a method on a union class, with one new wrinkle. In the variants that refer to the composite class (have fields of composite class type), computing `m` for embedded self references will usually involve delegating 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 HW4 and HW5, 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

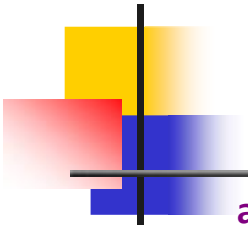
- An `IntList` is either:
  - `EmptyIntList()`, the empty list, or
  - `ConsIntList(first,rest)`, a non-empty list, where `first` is an `int` and `rest` is an `IntList`.
- Some examples include:
  - `EmptyIntList()`
  - `ConsIntList(7,EmptyIntList())`
  - `ConsIntList(12,ConsIntList(17,EmptyIntList()))`
- Java notation for constructed objects is wordy and awkward. Every explicit construction begins with the keyword **new**, *e.g.*,

```
new ConsIntList(12, new ConsIntList(17, new EmptyIntList()))
```



`IntList` Java Code

```
abstract class IntList { }

class EmptyIntList extends IntList { }

class ConsIntList extends IntList {
  private int first;
  private IntList rest;
  ConsIntList(int f, IntList r) {
    first = f;
    rest = r;
  }
  int first() { return first; }
  IntList rest() { return rest; }
}

/** The code in brown appears in full (standard) Java code.  It elided
  * in Functional Java, which is weakly supported in DrJava. */
```

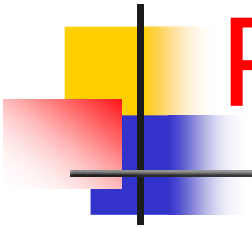


Defining Methods on IntList

```
abstract class IntList {
  abstract IntList sort() { }
}
class EmptyIntList extends IntList
  { IntList sort() { ... }
}
class ConsIntList extends IntList {
  private int first;
  private IntList rest;
  ConsIntList(int f, IntList r) {
     first = f;
     rest = r;
  }
  int first() { return first; }
  IntList rest() { return rest; }
  IntList sort() { ... }
}

/* Functional Java also auto-generates overrides of toString(),
    hashcode() and equals(...) supporting a functional (structural)
    interpretation of heap-allocated data.
```

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 {
  private int first;
  private IntList rest;
  ConsIntList(int f, IntList r) {
    first = f;
    rest = r;
  }
  int first() { return first; }
  IntList rest() { return 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));
  }
}
```



Four Important Idioms

Singleton Pattern

- 0-ary variants (no fields) in an inductive data definition typically have only one instance, *e.g.*, the empty list.

Strategy Pattern

- Functions as data values can be represented by instances of anonymous inner classes.

Visitor Pattern

- Encapsulate functions (closures) over a composite as separate objects (with methods containing the code for each variant) so new operations can be defined over a composite without modifying any classes in the composite.
- Perhaps the *most important idiom*

Parametric Polymorphism (Generic types)

- Classes (and methods) can be parameterized by type
- Regrettably type parameters are not first-class; in many contexts, generic types are either forbidden or ignored (with a warning message). As a result, the non-trivial use of generics in Java is messy, e.g. the generic version of HW5.

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`** (which means that the degenerate 0-ary constructor (auto-generated in Java) must be explicitly declared
- 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`**.

Nesting Variation

Entire composite can be embedded in the root class (interface?).

```
abstract class  IntList  {
  abstract public IntList sort();
  abstract public IntList insert(int i);
  public IntList cons(int i) { return new ConsIntList(i,this); }
  public IntList empty = new EmptyIntList(
  private static class EmptyIntList extends IntList {
    public IntList sort() { return this; }
    public IntList insert(int i) { return new ConsIntList(i, this); }
  }
  private static class ConsIntList extends IntList {
    private int first;
    private IntList rest;
    ConsIntList(int f, IntList r) {
      first = f;
      rest = r;
    }
    public int first() { return first; }
    public IntList rest() { return rest; }
    public IntList sort() { return rest.insert(first); }
    public IntList insert(int i) {
      if (i <= first) return new ConsIntList(i, this);
      else return new ConsIntList(first, rest.insert(i));
    }
  }
}
```

Strategy Pattern

- In Java 1.1 (the first revised release of Java), inner classes were added to the language. Static inner classes are really simple as we demonstrated in the previous coding example. They only change the visibility of raw class names and constructors and add class name prefixes (when visible) to the raw class; they have semantics identical to ordinary top-level classes. The interesting form of “inner class” is the “dynamic” inner class where every instance of such a class has an “enclosing instance”, which must be an instance of the enclosing class. In nearly all common usage, the enclosing class is the class of `this` (where the inner class is defined).

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, designed inner classes as the OO-analog of closures (the representation of a function value in a language supporting functions as data). Hence, an anonymous inner class is conceptually a closure like the closure representing an evaluated 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 an interface (or abstract class) for the particular function type we need. Stephen uses a library in Comp 310 with such interfaces (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);
}
```

  suffices.

- An anonymous inner class extends a type (typically an interface) filling in any method that is not yet defined and optionally overriding methods that are already defined. (I think fields are allowed also.) 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 …)

Visitor Pattern

- Given a inductively defined type represented using the composite pattern, how can we systematically define new operations over this type in OO style without modifying each class in the composite pattern representation, which is exactly what the interpreter pattern does.
- Let's use **`IntList`** as our example. There are only two variants in the composite representation: **EmptyIntList** and **ConsIntList**. How can we add new methods to the **`IntList`** composite without modifying the abstract class **`IntList`** or the concrete subclasses **EmptyIntList** and **ConsIntList**?
- We use two sneaky tricks:
  - Add a method to the top level class or interface and to each variant subclass called **`visit`**(<visitor>). Each concrete variant must define **`visit`**(<visitor>).
  - Define a visitor interface **`IntListVisitor`** corresponding to **`IntList`** that contains a **`forXXX`**(**`XXX host`**) method for each variant **`XXX`**.
- We put the code that we would have put in the definition of the new method in class **`XXX`** using the interpreter pattern as the body of the method **`forXXX`**(**`XXX host`**) with only one change: we replace all occurrence of **`this`** (both explicit and implied) by **`host`**. The host passed to **`forXXX`** inside the visitor is s simply **`this`** which is the composite object that is the conceptual receiver of the method call on the method implemented by the visitor. This pattern is sneaky but becomes natural with usage.

Visitor Pattern Example

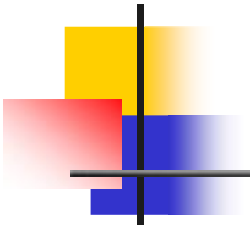
```
/** IntList := EmptyIntList + ConsIntList(int, IntList) */
abstract class List<T> extends Object {                          // extends Object is implied if omitted
  abstract public <R> R visit(ListVisitor<T,R> iv);                    // Visitor hook
  public ConsList cons(int i) { return new ConsIntList(i, this); }   // Adds i to the front of this.
  public static EmptyList<T> empty() { return EmptyList<T>.ONLY; }     // Returns the unique empty list.
}
class EmptyList<T> extends IntList {      // Concrete empty list class
  public static final EmptyList<T> ONLY = new EmptyList<T>();      // Singleton binding
  private EmptyList<T>() { }                                       // Single pattern private constructor
  public <R> R visit(ListVisitor<R,T> v) { return v.forEmptyList<T>(this); } // Visitor hook */
}
class ConsList<T> extends List<T> {       // Concrete non-empty list class
  T first;                               // element field
  List<T> rest;                            // rest field
  ConsIntList(int f, List<T> r) { first = f; rest = r; }   // Constructor for ConsList class
  public T first() { return first; }    // accessor for first
  public List<T> rest() { return rest; } // accessor for rest
  public <R> R visit(ListVisitor<T,R> v) { return v.forConsList(this); }   // Visitor hook
}
// For intelligible notation, need to override public String toString() inhertited from Object
```

Visitor Pattern Example cont

```
/** Visitor interface for the IntList type. */
interface ListVisitor<T,R> {
  abstract R forEmptyList(EmptyList<T> host);   // host replaces 'this' in Interpreter equivalent
  abstract R forConsList(ConsList<T> host);
}
class LengthVisitor<T> implements ListVisitor<T,Integer> {   // Visitor for length
  public static final LengthVisitor<T> ONLY = new LengthVisitor<T>(); // singleton binding
  private LengthVisitor<T>() { }                                      // singleton constructor
  public Object forEmptyList(EmptyList<T> el) { return 0; }
  public Object forConsIntList(ConsList<T> cil) { return 1 + cil.rest().visit(this); }
}
```

Java Conventions:

1. Every top-level "public" class *C* is stored in a separate file named *C*.**`java`**.
2. Visible top-level classes are "public", the only visibility we saw in Racket.
3. API top-level methods in Java are "public" and explicitly labeled as **`public`**, which is recognized by the compiler and tools (like junit) which requires many classes and methods to be labeled as **`public`**.
4. Empty package exceptions: the compiler relaxes it rules regarding "public" vs **`public`** in the empty package.



Generic (Parametric) Types

- Generic type parameters were added to Java 1.5 (relabeled as Java 5).
- Java generics are quirky because they were added after the Java Virtual Machine (JVM) was created and standardized. The instruction set of the JVM (called "bytecode") was designed to support Java 1.0 and all Java extensions since Java 1.0, including inner classes added in Java 1.1, are compiled by **`javac`** into the original JVM instruction set (JVM bytecode) which was designed prior to inner classes and generic types.
- The Java language committee at Sun Microsystems elected to use a translation of Java generic types called "erasure" (to my dismay).
- The only trace of generic types from your source code is embedded in the byte code as "annotations" (attributes ignored by the class loading process). Each generic field or method has an attribute declaring its generic type, so that Java source code with generics can be incrementally compiled (one class at a time, almost) assuming the compiled code (class files) for all references classes is available to the compiler. It uses these attributes to perform generic type checking.

Generic Types in Java Source Language

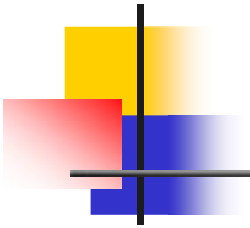
- Generic types can appear in most places where conventional types may appear with notable exceptions identified below.
- Occurrences of type variables must correspond to binding occurrences (just like local variables in Racket or Java). Binding occurrences are introduced in class/interface headers and method preludes.
- Generic class syntax

  ***modifiers* `class` *name* `<`*typeName*$_1$`, ..., `*typeName*$_n$`> { . . . }`**
- Generic (polymorphic) method syntax
- ***modifiers* `<`*typeName*$_1$`, ..., `*typeName*$_n$`>` *type methodName*`{ . . . }`**
- Simplification: *typeNames* may have upper bounds of form *extends type*
- Exceptions: The following operations are illegal when ***`T`*** is a type parameter

  naked array types, *e.g.* ***`T[]`***

  `new` operations of naked type, *e.g.* **`new T()`**

  casts to generic type (not just naked type variables)



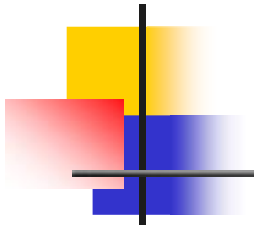
Pathologies of Java Generic Types

- The generic typing of arrays was built-in to Java 1.0 but array types were the *only* generic types. The subtyping rules for arrays are inconsistent with the rest of Java; Java supports array subtyping by performing run-time checks (using run-time type information!) that ensure type safety (data is never misinterpreted).
- In Java, the subtyping relation on generic types based on the values of type parameters is ugly and not supported by a run-time operation because the requisite information is not embedded in object representations.

Using Generic (Parametric) Types

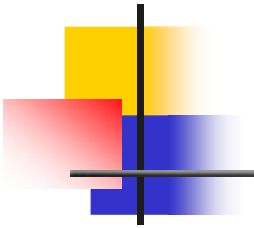
- For each composite data type we want a single visitor interface. To define methods of arbitrary return type, we must use class type parameters in the interface and method type parameters in the visit method hooks.
- Generically typed generalization of IntList visitor example.

```
interface IntListVisitor<T> {
  abstract T forEmptyIntList(EmptyIntList host);
  abstract T forConsIntList(ConsIntList host);
}

class LengthVisitor implements IntListVisitor<Integer> {
  public static final LengthVisitor ONLY = new LengthVisitor(); // singleton binding
  private LengthVisitor() { }                                   // singleton constructor

  public Integer forEmptyIntList(EmptyIntList el) { return 0; }
  public Integer forConsIntList(ConsIntList cil) { return 1 + cil.rest().visit(this); }
}
```



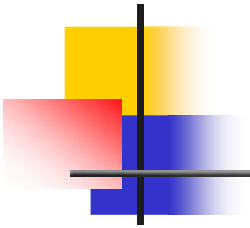
Visitor Demo Motivating Generics

Write a program that performs insertion sort on various types of lists where elements belong to an ordered type T (implementing Comparable<T>). How do you write a single definition for the insert method? How do you write it as a visitor?



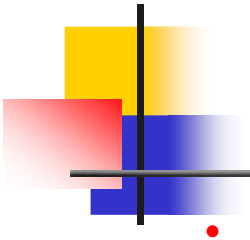
Parametric Polymorphism cont.

- In 1997-1998, programming language researchers proposed several different schemes 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 as too complex and a scheme based on type erasure proposed by a group including Phil Wadler and Martin Odersky won.
- 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 its 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 for personal reasons so Scala and Java both live with crippled type systems.



Parametric Polymorphism cont.

- The javac compiler enforces generic typing where generic types must match exactly (except for types with wildcards), 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 our Racket annotations) and back off (weakening your types and relying on occasional casts) when flagged for type errors reported by the compiler.
- Most important trick: use `ArrayList<T>` when you might want `T[]`. An `ArrayList` is simply an object wrapping (a pointer to) an array, which can be dynamically resized (unlike an array).
- 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 is signficant and complicates potential refactoring. It may be better to simply live with weaker typing. (Potential exception: widely used libraries. But usage is more difficult.)