CMSC 330: Organization of Programming Languages

Generics and Polymorphism

Polymorphism

- Definition
  - Feature that allows values of *different* data types to be handled using a uniform interface
- Applicable to
  - Functions
    - Same function applied to different data types
    - Example

```
let hd = function (h::_) -> h
```

  - Data types
    - Same data type can contain different data types
    - Example

```
type 'a option =
   None
 | Some of 'a
```


Two Kinds of Polymorphism

- Described by Strachey in 1967
- Ad hoc polymorphism
  - Range of types is finite
  - Combinations must be specified in advance
  - Behavior may *differ* based on type of arguments
- Parametric polymorphism
  - Code written without mention of specific type
  - May be transparently used with arbitrary number of types
  - Behavior is the *same* for different types of arguments

Polymorphism Overview

- Ad-hoc
  - Subtype (for object-oriented languages)
    - Sometimes not considered ad-hoc, but referred to as *subtype polymorphism*
  - Overloading, including operator overloading
- Parametric
  - ML types
  - Also known as generic programming (for object-oriented languages)
    - *Bounded parametric polymorphism* combines subtype and parametric polymorphism

Subtype Polymorphism

- Subtyping is a kind of polymorphism found in object-oriented programming languages, sometimes called *subtype polymorphism*
  - Allows a method to accept arguments of many types
  - Supported through inheritance
- Any function w/ object as parameter is polymorphic
  - If formal parameter is of class A, argument may be any object from a subclass of A

```
class A { … }
class B extends A { … }  // subclass
void f(A arg) { … }
A a= new A();
B b= new B();
f(a);  // f accepts argument of type A or B
f(b);
```


Liskov Substitution Principle

- Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.
  - I.e, if anyone expecting a T can be given an S, then S is a subtype of T.

Overloading

- Multiple copies of function, with the same function name but different numbers or types of parameters
- Arguments determine function actually invoked
  - Function is uniquely identified not by function name, but by name and order and number of argument type(s)
    - print(Integer i)→ print_Integer(…)
    - print(Float f) → print_Float(…)

```
static void print(Integer arg) { … }
static void print(Float arg)   { … }
print(1);      // invokes 1st print
print(3.14);   // invokes 2nd print
```

- It's an example of ad-hoc polymorphism

Operator Overloading

- Treat operators as functions with special syntax for invocation
  - Behavior different depending on operand type
- Example: + in Java

```
      1 + 2          // integer addition
    1.0 + 3.14       // float addition
"Hello" + "world"    // string concatenation
```

Operator Overloading (cont.)

- User-specified operator overloading
  - Supported in languages such as Ruby, C++
  - Makes user data types appear more like native types
- Examples defining a function for the ^ operator

```
class MyS
  def ^(arg)
    ...
  end
end
```

Ruby

```
class MyS {
  MyS operator^(MyS arg){
    ...
  }
}
```

C++

Parametric Polymorphism

- We saw *parametric polymorphism* in OCaml
  - It's polymorphism because polymorphic functions can be applied to many different types
- Found in statically typed functional languages such as OCaml, ML, Haskell
- Example:

```
let hd = function (h::_) -> h
```

'a list -> 'a

- Also used in object oriented programming
  - Known as *generic programming*
  - Example: Java, C++

A Stack of Integers

```
class IntegerStack {
  class Entry {
    Integer elt; Entry next;
    Entry(Integer i, Entry n) { elt = i; next = n; }
  }
  Entry theStack;
  void push(Integer i) {
    theStack = new Entry(i, theStack);
  }
  Integer pop() throws EmptyStackException {
    if (theStack == null)
      throw new EmptyStackException();
    else {
      Integer i = theStack.elt;
      theStack = theStack.next;
      return i;
    }
  }
}
```


IntegerStack Client

```
IntegerStack is = new IntegerStack();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = is.pop();
```

- This is OK, but what if we want other kinds of stacks?
  - Need to make one XStack for each kind of X
  - Problems: code bloat, maintainability nightmare

Polymorphism Using Object

```
class Stack {
  class Entry {
    Object elt; Entry next;
    Entry(Object i, Entry n) { elt = i; next = n; }
  }
  Entry theStack;
  void push(Object i) {
    theStack = new Entry(i, theStack);
  }
  Object pop() throws EmptyStackException {
    if (theStack == null)
      throw new EmptyStackException();
    else {
      Object i = theStack.elt;
      theStack = theStack.next;
      return i;
    }
  }
}
```


Stack Client

```
Stack is = new Stack();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = (Integer) is.pop();
```

- Now Stacks are reusable
  - push() works the same
  - But now pop() returns an Object
    - Have to downcast back to Integer, which is not checked until runtime

General Problem

- When we move from an X container to an Object container
  - Methods that take X's as input parameters are OK
    - If you're allowed to pass Object in, you can pass any X in
  - Methods that return X's as results require downcasts
    - You only get Objects out, which you need to cast down to X
- This is a general feature of subtype polymorphism

Parametric Polymorphism (for Classes)

- Starting in Java 1.5 we can *parameterize* the Stack class by its element type
- Syntax:
  - Class declaration: class A<T> { ... }
    - A is the class name, as before
    - T is a *type variable*, can be used in body of class (...)
  - Client usage declaration: A<Integer> x;
    - We *instantiate* A with the Integer type
  - Or A<String> y;

Parametric Polymorphism for Stack

```
class Stack<ElementType> {
  class Entry {
    ElementType elt; Entry next;
    Entry(ElementType i, Entry n) { elt = i; next = n; }
  }
  Entry theStack;
  void push(ElementType i) {
    theStack = new Entry(i, theStack);
  }
  ElementType pop() throws EmptyStackException {
    if (theStack == null)
      throw new EmptyStackException();
    else {
      ElementType i = theStack.elt;
      theStack = theStack.next;
      return i;
    }
  }
}
```


Stack<Element> Client

```
Stack<Integer> is = new Stack<Integer>();
Integer i;
is.push(new Integer(3));
is.push(new Integer(4));
i = is.pop();
```

- No downcasts
- Type-checked at compile time
- No need to duplicate Stack code for every usage

Parametric Polymorphism for Methods

- String is a subtype of Object
  1. static Object id(Object x) { return x; }
  2. static Object id(String x) { return x; }
  3. static String id(Object x) { return x; }
  4. static String id(String x) { return x; }

- Can't pass an Object to 2 or 4
- 3 doesn't type check
- Can pass a String to 1 but you get an Object back

Parametric Polymorphism, Again

- But id() doesn't care about the type of x
  - It works for *any* type

- So parameterize *the static method*:

  static <T> T id(T x) { return x; }

  Integer i = id(new Integer(3));

  - Notice no need to instantiate id; compiler figures out the correct type at usage

Standard Library, and Java 1.5 onward

- Generics in Java 1.5 came with a replacement for java.util.*
    - class LinkedList<A> { ...}
    - class HashMap<A, B> { ... }
    - interface Collection<A> { ... }

- But they didn’t change the JVM to add generics- how was that done?

Translation via Erasure

- Replace uses of type variables with Object

  class A<T> { ...T x;... } becomes

  class A { ...Object x;... }

- Add downcasts wherever necessary

  Integer x = A<Integer>.get(); becomes

  Integer x = (Integer) (A.get());

- So why did we bother with generics if they’re just going to be removed?
  - Because the compiler still did type checking for us
  - We know those casts won't fail at runtime

Limitations of Translation

- Some type information is not available at runtime
  - Recall type variables T are rewritten to Object

- Disallowed, assuming T is type variable
  - new T() would translate to new Object() (error)
  - new T[n] would translate to new Object[n] (warning)
  - Some casts/instanceofs that use T
    - (Only ones the compiler can figure out are allowed)

- Also produces some oddities
  - LinkedList<Integer>.class == LinkedList<String>.class
    - (These are uses of reflection to get the class object)

Using with Legacy Code

- Translation via type erasure
  - class A <T> becomes class A

- Thus class A is available as a “raw type”
  - class A<T> { ... }
  - class B { A x; } // use A as raw type

- Sometimes useful with legacy code, but...
  - It's a dangerous feature to use, plus unsafe
  - Relies on implementation of generics, not semantics