Overloading and (Parametric) Polymorphism

March 4, 2002

CS 131: Programming Languages

Overloading

- We will say that a function (or operator) is *overloaded* iff:
  - There is one name associated with several different definitions.
  - The context of the function's use determines which definition is used.
  - The decision is made by the compiler at compile time.

Examples

A Non-Example

```
let
   fun f1() = print "ah"
   fun f2() = print "oh"

   val g = if coinFlip() then f1 else f2
in
   g(); g(); g()
end
```

Another Non-Example

```
class Shape {
   ...
   void draw() {...defn 1...}
}
class Circle extends Shape
   int radius;
   void draw() {...defn 2...}
}
...
static void show(Shape s) {
   s.draw();
}
```

Code Reuse

- Type systems that are "too simple" can be overly restrictive.
  - e.g., quicksort in Pascal
- Some languages cope by using casts
  - e.g., sorting in Java or C

```
void qsort
   (void *base,  size_t nel, size_t width,
    int(*compar)(const void *, const void *));
```

Type Parameters

- Other languages allow variables ranging over *types* (i.e., whose values are types.)
  - Today we'll concentrate on C++, SML, and Generic Java
    `http://www.research.avayalabs.com/user/wadler/pizza/gj/`
- Two distinct but related uses
  - Generic types
  - Generic code

Generic Types in C++

```
template <class T> class Pair {
public:
   T fst;
   T second;
};

Pair<int> ip;
Pair<char*> csp;
```

Generic Types in SML

```
type 'a pair = 'a * 'a

val ip : int pair = (3,4)
val sp : string pair = ("a","b")
```

Generic Code in C++

```
template <class T>
void swap (T& x, T& y) {
  T t = x;
  x = y;
  y = t;
}

...
swap(intvar1,intvar2);
swap(stringvar1, stringvar2);
```

Generic Code in SML

```
fun 'a swap(x : 'a ref, y : 'a ref) : unit=
  let
     val t : 'a = !x
  in
     x := !y;
     y := t
  end
...
swap(intref1, intref2);
swap(boolref1, boolref2)
```

Generic Java

```
interface Iterator<A> {
   public A next();
   public boolean hasNext ();
}

interface Collection<A> {
   public void add(A x);
   public Iterator<A> iterator();
}
```

More Generic Java

```
class LinkedList<A> implements Collection<A> {
  // more stuff
  public void add(A elt) { ... }
}

// ...

LinkedList<String> ys =
    new LinkedList<string>();

ys.add("zero"); ys.add("one");
```

C++ vs GJ

```
template <class T>
bool mymax(T a, T b) {
  return (a>b ? a : b);
  }
```

```
template <class T>
bool mymax2(T a, T b){
 return (a.gt(b)?a:b);
}
```

```
static <T extends Number>
bool mymax(T a, T b) {
  return (a.doubleValue() > b.doubleValue() ?
            a : b);
}
```

Code Generation

- If we use some generic code (e.g., swap) on arguments of several types, does the program contain a single sequence of machine instructions for swap, or several?

A Puzzle

- Consider the definition

```
fun id x = x
```

- SML would say that

```
id : 'a -> 'a
```

meaning that, for any type 'a, if id is given a value of type 'a then the return value (if any) will also have the same type 'a.

```
val x : int*real = (id 3, id 4.0)
```

A Puzzle

- Now consider the definition

```
fun apply(f:'a ->'a, x:'a) = f(x)
```

- SML is perfectly happy to then compile

```
fun increment x = x+1
val y = apply(increment, 3)
```

even though the first argument to apply clearly isn't polymorphic. What's going on?

Solution

Subtyping vs. Polymorphism

- The identity function in Java

```
static Object id(Object x) {
  return x;
}
...
Object x = id(new Integer(0))
```

- The identity function in GJ

```
static <T> T id(T x) {
  return x;
}
...
Integer x = id(new Integer(0))
```

Subtyping *and* Polymorphism

```
static <T extends Number>
bool mymax(T a, T b) {
  return (a.doubleValue() > b.doubleValue() ?
            a : b);
}
```

Parametricity

- Consider SML without side-effects or infinite loops.
- Suppose f has type ∀'a. 'a->'a
  - What function could it be?

Parametricity

- Suppose f has type ∀'a. 'a list ->'a list
- What can f be?

Parametricity

- Informally, a polymorphic function is said to be *parametric* if its behavior is independent of its type argument.
  - i.e., same algorithm for all type instances.
- This can be elegantly formalized
  - "Related arguments yield related results"
  - But not in this class.
  - Application: TAL and callee-save registers