Polymorphism

Introduction

Example (Java)

```
int f (char a, char b) {
  return a==b;
}
```

Example (F#)

```
let f a b = (a = b);;
```

- Compare Java and F# function types
- The F# function is more flexible, since it can be applied to any pair of the same (equality-testable) type, Java to only **char** type.

Polymorphism

- Functions with that extra flexibility are called *polymorphic*
- A difficult word to define:
  - Applies to a wide variety of language features
  - Most languages have at least a little
  - We will examine four major examples, then return to the problem of finding a definition that covers them

Overloading

Overloading

- An *overloaded* function name or operator is one that has at least two definitions, all of different types
- For example, + and – operate on integers and reals
- Most languages have overloaded operators
- Some also allow the programmer to define new overloaded function names and operators

Predefined Overloaded Operators

Example (F#)

```
let x = 1 + 2;;
let y = 1.0 + 2.0;;
```

Example (Pascal)

```
a := 1 + 2;
b := 1.0 + 2.0;
c := "hello " + "there";
d := ['a'..'d'] + ['f']
```

Adding to Overloaded Operators

- Some languages, like C++, allow additional meanings to be defined for operators

Example (C++)

```
class complex {
  double rp, ip; // real part, imaginary part
public:
  complex(double r, double i) {rp=r; ip=i;}
  friend complex operator+ (complex,complex);
  friend complex operator* (complex,complex);
};

void f(complex a, complex b, complex c) {
  complex d = a + b * c;
  ...
}
```

Operator Overloading In C++

- C++ allows virtually all operators to be overloaded, including:
  - the usual operators $(+, -, *, /, \%, \text{'} \&, |, \ , !, =, <, >, +=, -=, =, *=, /=, \%=, \hat{}=, \&=, |=, <<, >>, >>=, <<=, ==, !=, <=, >=, \&\&, ||, ++, --, ->, *, ,)$
  - dereferencing (*p and p->x)
  - subscripting (a[i])
  - function call (f(a,b,c))
  - allocation and deallocation (**new** and **delete**)

Defining Overloaded Functions

- Some languages, like C++ or Java, permit overloading function names
- Must be unique function signature

Example

```
int square(int x) {
  return x*x;
}

double square(double x) {
  return x*x;
}
```

To Eliminate Overloading

- Could rename each overloaded definition uniquely
- Then rename each reference properly (depending on the parameter types)

Example

```
int square_i(int x) {
  return x*x;
}
double square_d(double x) {
  return x*x;
}

void f() {
  int a = square_i(3);
  double b = square_d(3.0);
}
```

Implementing Overloading

- Compilers usually implement overloading the same way:
  - Create a set of monomorphic functions, one for each definition
  - Invent a mangled name for each, encoding the type information
  - Have each reference use the appropriate mangled name, depending on the parameter types

Example: C++ Implementation

Example (C++)

```
int shazam(int a, int b) {return a+b;}
double shazam(double a, double b) {return a+b;}
```

Example (assembly result)

```
shazam__Fii:
        lda $30,-32($30)
        .frame $15,32,$26,0
        ...
shazam__Fdd:
        lda $30,-32($30)
        .frame $15,32,$26,0
        ...
```

Exercise

F# Result?

```
let f x = x + 1;;
let f x = x + 1.0;;
f 3;;
f 3.14;;
```

Java Result?

```
int f (int x) { return x + 1; }
double f (double x) { return x + 1.0; }
f (3);
f (3.14);
```

Parameter coercion

Coercion

- A *coercion* is an implicit type conversion, supplied automatically even when the programmer leaves it out

Example (Explicit)

```
double x;
x = (double) 2;
```

Example (Implicit)

```
double x;
x = 2;
```

Parameter Coercion

- Languages support different coercions in different contexts: assignments, other binary operations, unary operations, parameters…
- When a language supports coercion of parameters on a function call (or of operands when an operator is applied), the resulting function (or operator) is polymorphic

Example: Java

- This **f** can be called with any type of parameter Java is willing to coerce to type **double**

Example (Java)

```
void f(double x) { ... }

f((byte) 1);
f((short) 2);
f('a');
f(3);
f(4L);
f(5.6F);
```

Defining Coercions

- Language definitions take many pages of descriptions to define exactly which coercions are performed
- Some languages, especially older languages like Algol 68 and PL/I, have very extensive powers of coercion
- Some, like F#, have no coercion at all
- Most, like Java, are somewhere in the middle

Java Unary Numeric Promotion

(From Java Language Specification)

- Some operators apply unary numeric promotion to a single operand, which must produce a value of a numeric type:
  - If the operand is of compile-time type `Byte`, `Short`, `Character`, or `Integer`, it is subjected to unboxing conversion (§5.1.8). The result is then promoted to a value of type `int` by a widening primitive conversion (§5.1.2) or an identity conversion (§5.1.1).
  - Otherwise, if the operand is of compile-time type `Long`, `Float`, or `Double`, it is subjected to unboxing conversion (§5.1.8).
  - Otherwise, if the operand is of compile-time type `byte`, `short`, or `char`, it is promoted to a value of type `int` by a widening primitive conversion (§5.1.2).
  - Otherwise, a unary numeric operand remains as is and is not converted.
  - ...

Coercion and Overloading: Tricky Interactions

- Overloading uses the types to choose the definition
- Coercion uses the definition to choose a type conversion

Example

- Suppose that, like C++, a language is willing to coerce **char** to **int** or to **double**
- Which square gets called for: `square('a')`?

Example

```
int square(int x) {
  return x*x;
}
double square(double x) {
  return x*x;
}
```

Example Error

- Suppose that a language, such as C++, is willing to coerce **char** to **int**
- Which f gets called for: `f('a', 'b')`?
- Compiler error!

Example

```
void f(int x, char y) {
   ...
}
void f(char x, int y) {
   ...
}
```

Exercise

Results?

```
public class Ex1 {

  static double f ( double x ) { return x; }

  public static void main(String args[]) {
    System.out.println( f('a') );
    System.out.println( f((int) 97) );
    System.out.println( f(97L) );
    System.out.println( f(97.0) );
    System.out.println( f((float) 97.0) );
  }
}
```

Parametric polymorphism

Parametric polymorphism

- A function exhibits *parametric polymorphism* if it has a type that contains one or more type variables
- A type with type variables is a *polytype*
- Found in languages including F#, C++ and Ada

Example: C++ Function Templates

- Note that $>$ can be overloaded, so X is not limited to types for which $>$ is predefined.

Example

```
template <class X> X max(X a, X b) {
  return a > b ? a : b;
}

void g(int a, int b, char c, char d) {
  int m1 = max(a,b);
  char m2 = max(c,d);
}
```

Example: F# Functions

Example

```
> let identity x = x;;
val identity : x:'a -> 'a
> identity 3;; // returns 3
> identity "hello";; // returns "hello"
> let rec reverse L =
    match L with
      | [] -> []
      | h::t -> (reverse t)@[h];;
val reverse : L:'a list -> 'a list
> reverse [1;2;3];;
val it : int list = [3; 2; 1]
> reverse [(1,2); (3,4); (5,6)];;
val it : (int*int) list = [(5,6); (3,4); (1,2)]
```

Exercise

C++

```
template <class X> X max(X a, X b) {
  return a > b ? a : b;
}
double e = 1.0, f = 2.0;
int d1 = max(e, f);
double g[] = {1.0,2.0}, h[] = {2.0,1.0};
double i[] = max(g, h);
```

F#

```
let f x y = if x > y then x else y;;
f 3 4;;
f 3.0 4.0;;
```

Implementing Parametric Polymorphism

- One extreme: many copies
  - Create a set of monomorphic implementations, one for each type parameter the compiler sees
  - May create many similar copies of the code
  - Each one can be optimized for individual types
- The other extreme: one copy
  - Create one implementation, and use it for all
  - True universal polymorphism: only one copy
  - Can't be optimized for individual types
- Many variations in between

Subtype polymorphism

Subtype Polymorphism

- A function or operator exhibits *subtype polymorphism* if one or more of its parameter types have subtypes
- Important source of polymorphism in languages with a rich structure of subtypes
- Especially object-oriented languages: we'll see more when we look at C#

Example

Example (Pascal)

```
type
  Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
  Weekday = Mon..Fri;
function nextDay(D: Day): Day;
  begin
    if D=Sun then nextDay:=Mon else nextDay:=D+1;
  end;
procedure p(D: Day; W: Weekday);
  begin
    D := nextDay(D);
    D := nextDay(W);
  end;
```

Example

- A subtype of **Car** is **ManualCar**
- Variables reference objects of their type or subtype. They cannot reference (i.e. be assigned) supertype objects.
- Subtype objects can be substituted wherever supertype object is allowed.

Example (Java)

```
class Car {
  void brake() { ... }
}

class ManualCar extends Car {
  void clutch() { ... }
}
void g(Car z) { z.brake(); }
void h(ManualCar z) { z.brake(); }
void f(Car x, ManualCar y) {

  g(x);  // OK - x same type
  g(y);  // OK - y subtype
  h(x);  // Error - x supertype
  h(y);  // OK - y same type
}
```

Definitions and classifications

Polymorphism

- We have seen four kinds of polymorphic functions
  1. Overloading
  2. Parameter coercion
  3. Parametric polymorphism
  4. Subtype polymorphism
- There are many other uses of the word *polymorphic*:
  - Polymorphic variables, classes, packages, languages
  - Another name for runtime method dispatch: when **x.f()** may call different methods depending on the runtime class of the object **x**
  - Used in many other sciences
- No definition covers all these uses, except the basic Greek root: *many forms*

Definitions For Our Four

- A function or operator is polymorphic if it has at least two possible types
  - It exhibits *ad hoc* polymorphism if it has at least two but only finitely many possible types
  - It exhibits *universal* polymorphism if it has infinitely many possible types

Overloading

- An overloaded function name or operator is one that has at least two definitions, all of different types
- Ad hoc polymorphism
- Each different type requires a separate definition
- Only finitely many in a finite program

Parameter Coercion

- A *coercion* is an implicit type conversion, supplied automatically even if the programmer leaves it out
- Ad hoc polymorphism
- As long as there are only finitely many different types can be coerced to a given parameter type

Parametric Polymorphism

- A function exhibits *parametric* polymorphism if it has a type that contains one or more type variables
- Universal polymorphism
- As long as the universe over which type variables are instantiated is infinite

Subtype Polymorphism

- A function or operator exhibits *subtype* polymorphism if one or more of its parameter types have subtypes
- Universal polymorphism
- As long as there is no limit to the number of different subtypes that can be declared for a given type
- True for all class-based object-oriented languages, like Java/C#