Polymorphism

Introduction

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

C++:

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

F#:

```
let f a b = (a = b);;
  val f : a:'a -> b:'a -> bool
```

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

Outline

- Overloading
- Parameter coercion
- Parametric polymorphism
- Subtype polymorphism
- Definitions and classifications

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

F#:
```
let x = 1 + 2;;
let y = 1.0 + 2.0;;
```

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

```
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 (`+,-,*,/,%,^,&,|,~,!,=,<,>, +=,-=,=,*=,/=,%=,^=,&=,|=,<<,>>,>>=,<<=,==, !=,<=,>=,&&,||,++,--,->,*,,`)
  - dereferencing (`*p` and `p->x`)
  - subscripting (`a[i]`)
  - function call (`f(a,b,c)`)
  - allocation and deallocation (`new` and `delete`)

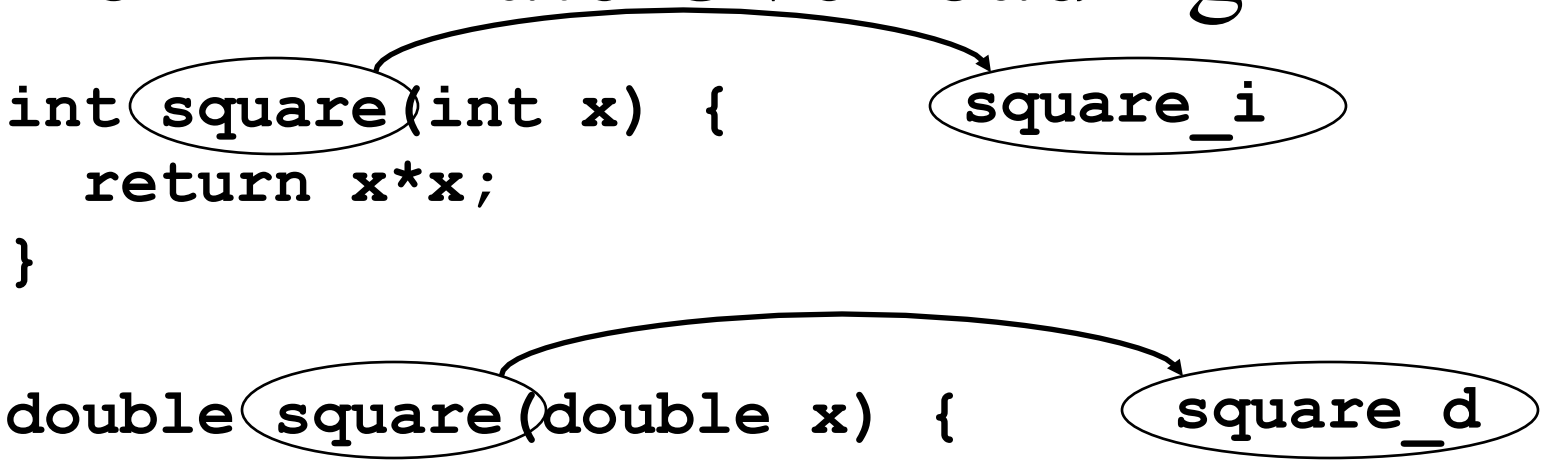
Defining Overloaded Functions

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

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

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

To Eliminate Overloading



Could rename each overloaded definition uniquely…

How To Eliminate Overloading

```
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);
}
```

Then rename each reference properly (depending on the parameter types)

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

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

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

Compiler:

`shazam(3, 4);` generates call to **`shazam__Fii`**

`shazam(3.0, 4.0);` generates call to **`shazam__Fdd`**

Exercise 0

1. What does F# do with:

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

2. What about Java?

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

3. What can you say about overloading for F# and Java?

Outline

- Overloading
- Parameter coercion
- Parametric polymorphism
- Subtype polymorphism
- Definitions and classifications

Coercion

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

Explicit type conversion in Java:

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

Coercion (implicit conversion) in Java:

```
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

```
void f(double x) {
   …
}

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

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

Defining Coercions

- Language definitions complex to define exactly which coercions are performed
- Some languages, especially some older languages like Algol 68 and PL/I, have very extensive powers of coercion
- Some, like F#, have *none*
- Most, like Java, are somewhere in the middle

Example: Java Unary Promotion

5.6.1 Unary Numeric Promotion

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`**, or **`char`**, unary numeric promotion promotes it to a value of type **`int`** by a widening conversion (§5.1.2). Otherwise, a unary numeric operand remains as is and is not converted.

Unary numeric *promotion* is performed on expressions in the following situations: the dimension expression in array creations (§15.9); the index expression in array access expressions (§15.12); operands of the unary operators plus **`+`** (§15.14.3) and minus **`-`** (§15.14.4) ...

The Java Language Specification
James Gosling, Bill Joy, Guy Steele

Coercion and Overloading: Tricky Interactions

- There are potentially tricky interactions between overloading and coercion
  - 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')` ?

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

*Example

- Suppose that a language, such as C++, is willing to coerce **`char`** to **`int`**
- Which **`f`** gets called for:

  `f('a', 'b')` ?

  Compiler error on .NET C++, cannot determine which function to call.

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

Exercise 1

- What is the result of the following C++?

```
#include "stdafx.h"
#include <iostream.h>

double f (double x) { return x;}

void main(void)
{     cout << f('a‘) ;
      cout << f((int) 97);
      cout << f(97L);
      cout << f(97.0);
      cout << f((float) 97.0);
}
```

Exercise 1 Continued

- What is the result of the following Java?

```
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) );
     }
}
```

Outline

- Overloading
- Parameter coercion
- Parametric polymorphism
- Subtype polymorphism
- Definitions and classifications

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.*

```
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);
}
```

```
int max(int a, int b) {
  return a > b ? a : b;
}
```

```
char max(char a, char b) {
  return a > b ? a : b;
}
```

Example: F# Functions

a)
```
let identity x = x;;
  val identity : x:'a -> 'a

identity 3;;
 val it : int = 3

identity "hello";;
 val it : string = "hello"
```

b)
```
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 1.1

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

What happens and why?

1.
```
double e=1.0, f=2.0;
int d1 = max( e, f);
```
2.
```
double g[]={1.0,2.0}, h[]={2.0,1.0};
double i[] = max( g, h);
```
3.
```
let f x y = if x > y then x else y;;
```
4.
```
f 3 4;;
```
5.
```
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

Outline

- Overloading
- Parameter coercion
- Parametric polymorphism
- Subtype polymorphism
- Definitions and classifications

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: 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;
```

Subtype polymorphism: **`nextDay`** *can be called with a subtype parameter*

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

}
```

A subtype of **`Car`** *is* **`ManualCar`**

Variables and subtypes

Variables reference objects of their type or subtype. Cannot reference (i.e. be assigned) supertype objects.

Substitution

Subtype objects can be substituted wherever supertype object allowed.

More Later

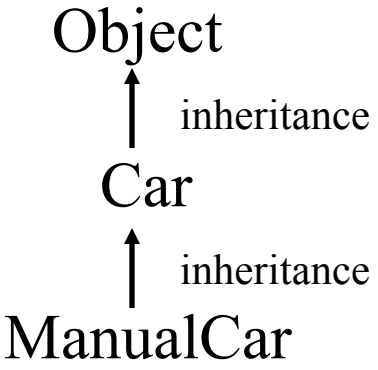
- We'll see more about subtype polymorphism when we look at object-oriented languages

Exercise 1.3

Are the following valid? Why?

```
void f() {
     Car c;
     ManualCar m;

1.   c = m;
2.   m = c;
3.   m = (ManualCar) c;
```




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

class ManualCar extends Car {
  void clutch() { … }
}
```

Outline

- Overloading
- Parameter coercion
- Parametric polymorphism
- Subtype polymorphism
- 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 *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: *many forms*

Definitions For Our Four

A function or operator is *polymorphic* if it has at least two possible types

1. It exhibits *ad hoc polymorphism* if it has at least two but only finitely many possible types
2. It exhibits *universal polymorphism* if it has infinitely many possible types

Overloading

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

Parameter Coercion

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

Parametric Polymorphism

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

Subtype Polymorphism

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