

Polymorphism

Introduction

- Compare these function types
- The ML function is more flexible, since it can be applied to any pair of the same (equality-testable) type

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

```
ML:     - fun f(a, b) = (a = b);  
        val f = fn : 'a * '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
- Many languages have overloaded operators
- Some also allow the programmer to define new overloaded function names and operators

Predefined Overloaded Operators

```
ML:  val x = 1 + 2;  
     val 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++, permit the programmer to overload function names

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

To Eliminate Overloading

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

A curved arrow points from the text `square(int x)` to the text `square_i`, which is enclosed in an oval.

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

A curved arrow points from the text `square(double x)` to the text `square_d`, which is enclosed in an oval.

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

You 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:  
            Ida $30, -32($30)  
            .frame $15, 32, $26, 0  
            ...  
            shazam__Fdd:  
            Ida $30, -32($30)  
            .frame $15, 32, $26, 0  
            ...
```

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 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);
```

This **f** can be called with any type

```
f((short) 2);
```

of parameter Java is willing to

```
f('a');
```

coerce to type **double**

```
f(3);
```

```
f(4L);
```

```
f(5.6F);
```

Defining Coercions

- Language definitions often take many pages 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 ML, have none
- Most, like Java, are somewhere in the middle

Example: Java

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, like C++, a language is willing to coerce **char** to **int**
- Which **f** gets called for **f('a', 'b')** ?

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

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 ML, C++ and Ada

Example: C++ Function Templates

```
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);  
}
```

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

Example: ML Functions

```
- fun identity x = x;  
val identity = fn : 'a -> 'a  
- identity 3;  
val it = 3 : int  
- identity "hello";  
val it = "hello" : string  
- fun reverse x =  
  = if null x then nil  
  = else (reverse (tl x)) @ [(hd x)];  
val reverse = fn : 'a list -> 'a list
```

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 Java

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() { ... }  
}
```

*A subtype of **Car** is*

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

ManualCar

*Function **g** has an unlimited number of types—one for every class we define that is a subtype of **Car***

```
void g(Car z) {  
    z.brake();  
}
```

```
void f(Car x, ManualCar y) {  
    g(x);  
    g(y);  
}
```

That's subtype polymorphism

More Later

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

Outline

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

Polymorphism

- We have seen four kinds of polymorphic functions
- 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*
- Here are definitions that cover our four...

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

- Ad hoc polymorphism
- Each different type requires a separate definition
- Only finitely many in a finite program

Parameter Coercion

- Ad hoc polymorphism
- As long as there are only finitely many different types can be coerced to a given parameter type

Parametric Polymorphism

- Universal polymorphism
- As long as the universe over which type variables are instantiated is infinite

Subtype Polymorphism

- Universal
- 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