Chapter 8 Polymorphism 多态性

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
```

- Functions with that extra flexibility are called *polymorphic*

重载函数

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

3

Predefined Overloaded Operators

可以连接两种以上的数据类型.

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']
```

4

Adding to Overloaded Operators

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

实部 虚部

```
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;
  …
}
```

友元函数

5

Operator Overloading in C++

- C++ allows virtually all operators to be overloaded, including: 允许几乎所有运算符重载
  - the usual operators (**+**,**-**,*****,**/**,**%**,**^**,**&**,**|**,**~**,**!**,**=**,**<**,**>**, **+=**,**-=**,**=**,***=**,**/=**,**%=**,**^=**,**&=**,**|=**,**<<**,**>>**,**>>=**,**<<=**,**==**, **!=**,**<=**,**>=**,**&&**,**||**,**++**,**--**,**->***,**,**)
  - dereferencing (`*p` and `p->x`)
  - subscripting (`a[i]`)
  - function call (`f(a,b,c)`)
  - allocation and deallocation (`new` and `delete`)

6

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;
}
```

7

To Eliminate Overloading

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

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

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

You could rename each overloaded definition uniquely…

8

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 in that same way:
  - Create a set of monomorphic functions, one for each definition 创建一组单态函数
  - 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
        …
```

11

Parameter Coercion 强制转换

- 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;
```

12

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 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

15

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

16

Example

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

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

→ int '的'

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) {
  …
}
```

Example

- Consider an unknown language with integer and real types in which **`1+2`**, **`1.0+2`**, **`1+2.0`** and **`1.0+2.0`** are all legal expressions.
  - Result of coercion without overloading:
    The operator applies only to pairs of real numbers. All integers are coerced to real before the operator is applied.
  - Result of overloading without coercion:
    The operator has four types: `int*int->int`, `int*real->real`, `real*int->real` and `real*real->real`. No coercion is performed.

只要有一个real,强制转换成real*real→real作用,4种类型只需2种类型就够

Example

- Consider an unknown language with integer and real types in which **`1+2`**, **`1.0+2`**, **`1+2.0`** and **`1.0+2.0`** are all legal expressions.
  - Result of coercion and overloading:
    The operator has two types: `int*int->int` and `real*real->real`. When the operands are of mixed types, the `int` is coerced to `real` before applying the `real*real->real` operator.

参数化多态，

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++, Ada, and Java

21

Example: C++ Function Templates

Using a C++ template to define a polymorphic function called **max**.

```
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);  // int * int -> int
  char m2 = max(c,d); // char * char -> char
}
```

type variable

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

22

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

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

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

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

Implementing Parametric Polymorphism

- 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

25

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

26

Example: Pascal

A subtype of the type **`Day`**

```
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*

27

Example: Java

Define a Java class called **`Car`** with a function called **`break`**.

`ManualCar` is a subtype of **`Car`**.

Define a Java class called **`ManualCar`** with an extra function called **`clutch`**.

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

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

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

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

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

That's subtype polymorphism

28

Example

- Consider an unknown language with integer and real types in which **`1+2`**, **`1.0+2`**, **`1+2.0`** and **`1.0+2.0`** are all legal expressions.
  - Result of subtype polymorphism, with no overloading or coercion:
    The language treats `int` as a subtype of `real`: an `int` is a real numbers for which the fractional part is zero. It uses the same representation for both. There is a single multiplication operator and type conversion at runtime.

29

Questions?

30