Linguagens de Programação

Polimorfismo

Andrei Rimsa Álvares
andrei@cefetmg.br

Sumário

- Introdução
- Polimorfismo
  - Sobrecarga
  - Coerção
  - Inclusão/Subtipo
  - Paramétrico

INTRODUÇÃO

Linguagens de Programação

Introdução

- Considere as seguintes funções em C e ML que realizam comparações de dois valores

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

Em C

```
fun f (a, b) = (a = b);
```

Em ML

Qual a diferença entre essas duas funções?

Introdução

- A diferença principal é que a função em C é somente aplicada a caracteres, enquanto a função em ML pode ser aplicada a qualquer par de mesmo tipo (desde que comparável por igualdade)

```
- fun f (a, b) = (a = b);
stdIn:1.20 Warning: calling polyEqual
val f = fn : ''a * ''a -> bool
```

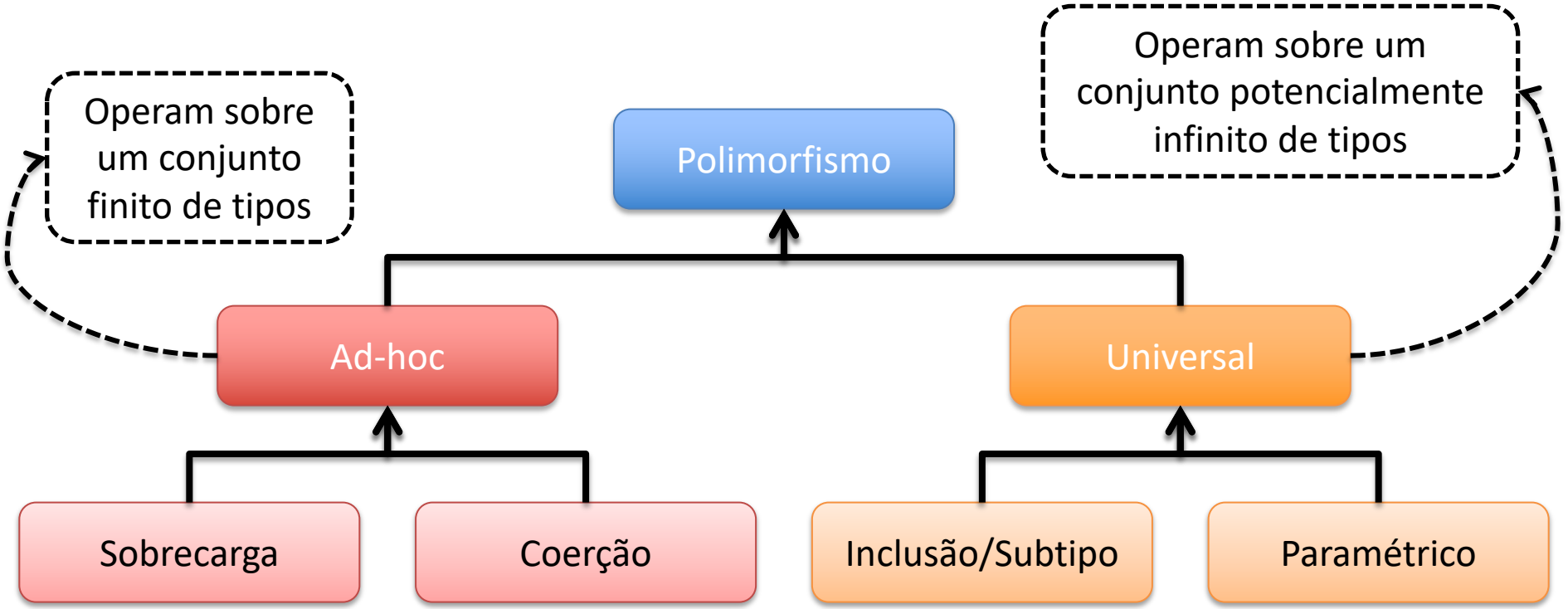
  - Funções com este tipo de flexibilidade extra são chamadas de polimórficas (do grego, múltiplas formas)

POLIMORFISMO

Linguagens de Programação

Polimorfismo

- Os tipos de polimorfismos podem ser organizados de acordo com a **classificação de Cardelli**



POLIMORFISMO > SOBRECARGA

Linguagens de Programação

Polimorfismo Ad-hoc: Sobrecarga

- Um nome de função ou operador **sobrecarregado** é tal que possui pelo menos duas definições, todas com tipos diferentes

- Muitas linguagens possuem operadores sobrecarregados
  - Como em ML com o operador + para soma de inteiros e reais

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

  - Ou em Pascal que o operador +, além de somar inteiros e reais, pode ser usado para concatenar strings e fazer uniões de conjuntos

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

Polimorfismo Ad-hoc: Sobrecarga

- Outras linguagens permitem ao programador adicionar definições para operadores já existentes
  - Como em C++ que permite sobrecarregar operadores de adição (+) e multiplicação (*), *shift left* (<<)

```
class Complex {
  double rp;
  double ip;
public:
  Complex(double rp, double ip) : rp(rp), ip(ip) {}
  friend Complex operator+(const Complex& x, const Complex& y);
  friend Complex operator*(const Complex& x, const Complex& y);
  friend std::ostream& operator<<(std::ostream& output, const Complex& c);
};
```

```
Complex f(Complex a, Complex b, Complex c) {
  Complex d = a + b * c;
  return d;
}
```

Quais outros operadores em C++ podem ser sobrecarregados?

Polimorfismo Ad-hoc: Sobrecarga

- Algumas linguagens, como C++, permitem que programadores sobrecarreguem nomes de funções

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

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

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

A linguagem C não permite tal sobrecarga, como este código pode ser reescrito nesta linguagem?

Polimorfismo Ad-hoc: Sobrecarga

- Em C, deve-se criar um nome único para cada função e renomear cada chamada de acordo

```
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);
}
```

Como que o código anterior é compilado com os nomes de funções sobrecarregados?

Polimorfismo Ad-hoc: Sobrecarga

- O sistema de linguagem de C++ usa uma abordagem similar
  1. Cria-se um conjunto de funções monomórficas, uma para cada definição usando um esquema de codificação (*mangling*) que inclui informações de tipos

```
$ objdump -t square | grep square
000000000000072c g     F .text0000000000000018              _Z6squared
0000000000000714 g     F .text0000000000000018              _Z6squarei
$ c++filt _Z6squared _Z6squarei
square(double)
square(int)
```

  2. Para cada invocação, use o nome (*mangled*) apropriado de acordo com o tipo do parâmetro

```
$ objdump --demangle --disassemble='f()' test2
...
0000000000000744 <f()>:
 ...
 750:    97fffff1     bl   714 <square(int)>
 ...
 75c:    97fffff4     bl   72c <square(double)>
```

POLIMORFISMO > COERÇÃO

Polimorfismo Ad-hoc: Coerção

- Uma conversão de tipos implícita é chamada de coerção
  - Exemplo de conversão implícita em Java (coerção)

```
double x = 2;
```

Java converte automaticamente o inteiro `2` em `2.0` (`double`)

  - O resultado é equivalente a ter feito uma conversão explícita

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

Polimorfismo Ad-hoc: Coerção

- Linguagens suportam diferentes coerções em diferentes contextos
  - Como em atribuições, operações binárias, unárias, parâmetros, etc

- Quando aplicada a parâmetros em uma chamada de funções (ou quando aplicada a um operador), a função resultante (ou operador) é polimórfica

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

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

Java faz a coerção destes tipos (`byte`, `short`, `char`, `int`, `long` e `float`) para `double`

Coerção vs Sobrecarga

- Definições de linguagens com frequência usam muitas páginas para definir exatamente quais coerções são feitas
  - Algumas linguagens antigas como Algol 68 e PL/I possuem poder extenso de coerções
  - Outras como ML, não possuem nenhum tipo de coerção
  - Já a maioria das linguagens, como Java, situam-se no meio do caminho

Coerção vs Sobrecarga

- Existem potenciais riscos entre as interações de sobrecarga e coerção
  - Sobrecarga usa os tipos para escolher a definição
  - Coerção usa definições para escolher a conversão de tipos

- Considere os trechos de códigos a seguir em C++

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

Qual dessas funções é usada na chamada `square('a')`?

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

E nesse caso para a chamada `f('a', 'b')`?

POLIMORFISMO > INCLUSÃO/SUBTIPO

Linguagens de Programação

Polimorfismo Universal: Inclusão/Subtipo

- Como visto anteriormente, um subtipo é um subconjunto
  - Se o parâmetro de uma função funciona com um conjunto, ela também funciona com um subconjunto deste conjunto
- É natural que linguagens permitam que uma função com parâmetro de tipo `t` possa ser chamada com um subtipo de `t`

```
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(var D: Day;
                W: Weekday);
  begin
    D := nextDay(D);
    D := nextDay(W)
  end;
```

Polimorfismo Universal: Inclusão/Subtipo

- Uma função ou operador exibe polimorfismo de subtipo se um ou mais de seus tipos de parâmetros possuem subtipos
  - Polimorfismo de subtipos surge automaticamente em linguagens com subtipos (como Pascal)
  - Uma coleção muito mais rica de subtipos ocorre em linguagens orientadas a objetos (como Java)

Polimorfismo Universal: Inclusão/Subtipo

- Suponha um carro que responde ao pressionamento dos freios, portanto pode-se criar a classe `Car` com método `brake`

```
class Car {
  public void brake() { /* ... */ }
}
```

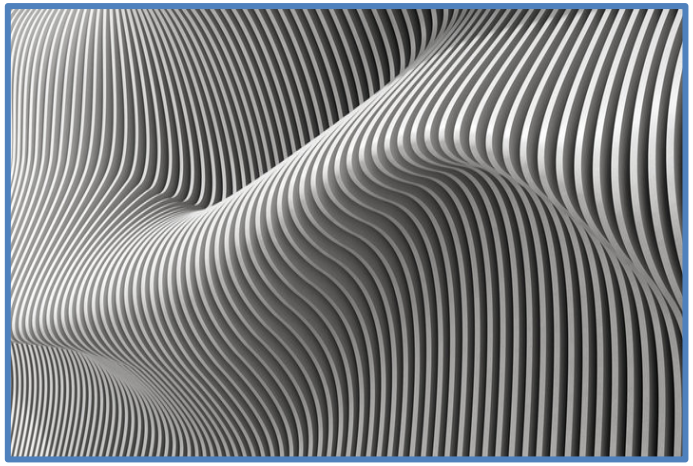
- Alguns carros possuem outras funcionalidades, como marcha manual, portanto pode-se criar a classe `ManualCar` com método `clutch` (o uso de `extends` indica que `ManualCar` é um subtipo de `Car`)

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

- Agora qualquer método (ex.: `g`) que recebe `Car` como parâmetro é automaticamente polimórfico, ou seja, também aceita `ManualCar`

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

POLIMORFISMO > PARAMÉTRICO



Polimorfismo Universal: Paramétrico

- Considere a seguinte função polimórfica em ML

```
- fun f(a, b) = (a = b);
stdIn:1.19 Warning: calling polyEqual
val f = fn : ''a * ''a -> bool
```

  - O *tipo variável* `''a` pode ser substituído por qualquer tipo testável por igualdade (`int`, `char`, ...)
  - Este tipo de polimorfismo é chamado de polimorfismo paramétrico

- Uma função exibe polimorfismo paramétrico se seu tipo contém um ou mais **tipos variáveis** (como `''a * ''a -> bool`)
  - Um tipo com tipos variáveis é chamado de *polytype*

Polimorfismo Universal: Paramétrico

- Polimorfismo paramétrico é encontrado em várias linguagens, como no exemplo a seguir em C++

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

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

Como C++ implementa este tipo de polimorfismo paramétrico?

  - A função polimórfica `max` usa *templates* de C++ em sua definição, tal que `X` é um *tipo variável*

Cuidado: a variável `X` não funciona com qualquer tipo, mas com aqueles cuja operação de comparação (>) é definida

Dica: como C++ pode sobrecarregar o operador >, `X` não está limitado a apenas os tipos que possuem este operador predefinido

Polimorfismo Universal: Paramétrico

- C++ utiliza a estratégia de criar múltiplas cópias separadas para a função paramétrica, uma para cada tipo utilizado
  - Para o exemplo anterior, o compilador de C++ percebe que a função `max` foi usada com `X = int` e `X = char` e cria estas duas definições (sobrecarregadas)

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

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

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

Qual a vantagem e desvantagem desta abordagem?

Polimorfismo Universal: Paramétrico

- Java também suporta polimorfismo paramétrico usando *Generics*, como no exemplo a seguir

```
public static <T> T head(T[] array) {
    return array[0];
}

public static void g() {
    String a[] = { "Hello", "world" };
    String s = head(a);

    Integer x[] = { 1, 2, 3 };
    int y = head(x);
}
```

Como Java implementa este tipo de polimorfismo paramétrico?

Polimorfismo Universal: Paramétrico

- Java utiliza a estratégia de criar uma única cópia desta função com um tipo genérico limitado
  - No caso do exemplo anterior, substitui-se o tipo `T` por `Object` e insere conversões de tipos para preservar a segurança de tipos

```
public static Object head(Object[] array) {
    return array[0];
}

public static void g() {
    String[] a = { "Hello", "world" };
    String s = (String) head(a);

    Integer[] x = { 1, 2, 3 };
    int y = (Integer) head(x);
}
```

Este algoritmo é conhecido como **Type Erasure**

Qual a vantagem e desvantagem desta abordagem?

ISSO É TUDO, PESSOAL!