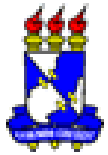
Polimorfismo

Prof. Alberto Costa Neto
alberto@ufs.br

Linguagens de Programação



Departamento de Computação
Universidade Federal de Sergipe

Preliminares

- Linguagens tradicionais possuem sistema de tipos simples
  - constantes, variáveis, resultado de funções, parâmetros formais, etc: *devem ser declarados com um tipo específico*
- Essas linguagens são chamadas de Monomórficas

Monomorfismo

- Sistema de tipos puramente monomórfico implica em algumas limitações
  - Reuso fica comprometido
  - Requer representação e operações distintas para cada tipo de elemento

```
type ConjuntoChar = set of Char
function disjunto(s1, s2: ConjuntoChar):Boolean;
begin
   disjunto := (s1*s2 = [])
end
```

em Pascal

nenhuma operação particular a caracteres, porém...

...função é monomórfica do tipo:

ρ Caracter × ρ Caracter → Valor-Verdade

Monomorfismo

- Muitos algoritmos e estruturas de dados são inerentemente genéricos
  - Exemplos:
    - Operações sobre elementos de Conjuntos são totalmente independentes do tipo dos elementos
    - Algoritmo para ordenação independe do tipo do elemento a ser ordenado (desde que seu tipo possua operação de comparação)
    - Uma lista ou árvore deveria armazenar qualquer tipo de dado

Polimorfismo

- Capacidade de um único identificador poder denotar entidades de tipos diversos
  - Ex: Diferentes funções com mesmo nome e que diferenciam-se pelos tipos parâmetros
- Em geral, o polimorfismo em linguagens tipadas tem sido introduzido como um mecanismo para reuso (código genérico)

Classificação de Polimorfismo

- Tipos:

Ad-hoc	*Coerção*
	Sobrecarga (Overloading)
Universal	*Paramétrico*
	Inclusão

Reuso x Polimorfismo

- Ad-hoc
  - Aparentemente proporciona reuso, mas existe uma implementação para cada tipo admissível
- Universal
  - Uma mesma implementação pode atuar sobre elementos de tipos diferentes

Ad-hoc::Coerção

- Mapeamento implícito de valores de um tipo para valores de um outro tipo
- Executada automaticamente pelo compilador sempre que necessário

```
void f (float i) { }
main() {
    long num;
    f (num);
}
```

em C

funciona? Sim!

f lida com *float* e *long*? Não!

Compilador se encarrega de embutir código para transformar *long* em *float*

  - C possui tabela de conversões permitidas

Ad-hoc::Coerção

- Ampliação
  - Tipo de menor conjunto de valores para tipo de maior conjunto: Operação segura pois valor do tipo menor necessariamente tem correspondente no tipo maior

- Estreitameno
  - Tipo de maior conjunto de valores para tipo de menor conjunto: Operação insegura pois pode haver perda de informação

Ad-hoc::Coerção

- Redigibilidade?
  - Não demanda chamada de funções explícitas de conversão de tipos
- Confiabilidade?
  - Pode impedir a detecção de certos tipos de erros

em C

```
main() {
    int a, b = 2, c = 3;
    float d;
    d = a * b;


    a = b * d;
}
```

Maior redigibilidade que fazer d = (float) (a*b);

Menor confiabilidade:
O programador queria digitar a = b*c; (errou)
Mas o compilador não dá erro!

Ad-hoc::Coerção

- ADA não admite coerções
- Java só admite coerção por ampliação

```
byte a, b = 10, c = 10;
int d;

d = b;

c = d;
```

em Java

menor → maior:
coerção por ampliação

maior → menor:
uso de type casts

(byte)

Coerção nas LP

- C é muito permissiva com relação a coerções
- ADA e Modula-2 não admitem coerções
- Java busca o meio termo, permitindo apenas coerções com ampliações

Ad-hoc::Sobrecarga

- Um mesmo identificador (ou operador) é usado para designar duas ou mais operações distintas
- Aceitável quando o uso do identificador (ou operador) não é ambíguo: uso de assinaturas diferentes

Ad-hoc::Sobrecarga

- Exemplo: Operador "–" de Pascal denota
  - Negações para tipos numéricos distintos

    (Integer → Integer) ou **(Real → Real)**
  - Subtrações para tipos numéricos distintos

    (Integer × Integer → Integer)

    ou **(Real × Real → Real)**
  - Subtrações entre conjuntos

    (Set × Set → Set)

Ad-hoc::Sobrecarga

- Em C:
  - Embute sobrecarga em seus operadores
  - Não se pode implementar novas sobrecargas de operadores
  - Não existe qualquer sobrecarga de subprogramas
- Em Pascal
  - Existem subprogramas sobrecarregados na biblioteca padrão: ex: *read* e *write*
  - Não se pode implementar novas sobrecargas de subprogramas

Ad-hoc::Sobrecarga

- Em Java
  - Embute sobrecarga em operadores e em subprogramas de suas bibliotecas
  - Apenas subprogramas podem ser sobrecarregados pelo programador
- Em ADA e C++
  - Realizam e permitem que programadores realizem sobrecarga de subprogramas e operadores

Ad-hoc::Sobrecarga

em C++

```
class ExOp {
        int v;
public:
        ExOp() { v = 0; }
        ExOp(int j) { v = j; }
        ExOP operator +(const ExOp& u) const {
           return ExOp(v + u.v);  }
        ExOP& operator +=(const ExOp& u) {
           v += u.v; return *this; }
};
main() {
        int a = 1, b = 2, c = 3;
        c += a;
        ExOp r(1), s(2), t;
        r += s;
        t = r + s;
}
```

Ad-hoc::Sobrecarga

- Tipos de Sobrecarga
  - **Independente de Contexto**
    - Lista de parâmetros deve ser distinta na quantidade ou no tipo dos parâmetros
    - Tipo de retorno **não pode** ser usado para diferenciação
    - Ex: Pascal, ML, JAVA, C++
  - **Dependente de Contexto**
    - Tipo de retorno pode ser usado para diferenciação
    - Mais esforço do compilador: precisa fazer análise do contexto
    - Pode provocar erros de ambigüidade
    - Ex: ADA

Ad-hoc::Sobrecarga

- Independente de Contexto

```
void f(void) { }
void f(float) { }
void f(int, int) { }
void f(float, float) { }
int f(void) { }
main() {
    f();
    f(2.3);
    f(4, 5);
    f(2.2f, 7.3f);
}
```

não pode ser diferenciada pelo tipo de retorno

Ad-hoc::Sobrecarga

- Dependente de Contexto
- Em ADA: operador "/ " designa
  - divisão real: **float × float → float**
  - divisão inteira: **integer × integer → integer**
- Sobrecarga de "/ " :

```
function "/" (m,n : integer) return float is
begin
   return float (m) / float (n);
end "/";
```

em ADA

integer × integer → float

Ad-hoc::Sobrecarga

- Dependente de Contexto

em ADA

```
n : integer;
x : float;
...
x: = 7.0/2.0;          -- calcula 7.0/2.0 = 3.5

x: = 7/2;              -- calcula 7/2 = 3.5

n: = 7/2;              -- calcula 7/2 = 3

n: = (7/2)/(5/2); -- calcula (7/2)/(5/2) = 3/2 = 1

x: = (7/2)/(5/2); --
```

ambiguidade (pode ser 1.4 ou 1.5)

Sugestões de Leitura

- Concepts of Programming Languages (Robert Sebesta)
  - Capítulo 9.8 a 9.11
- Programming Language Concepts and Paradigms (David Watt)
  - Seções 8.3 e 8.4
- Linguagens de Programação (Flávio Varejão)
  - Seções 7.1.4 e 7.2