# Sistemas de Tipos

Sérgio Queiroz de Medeiros
sergio@ufs.br

29 de março de 2012

# Tipos pra quê?

- Tipos fornecem um contexto implícito para muitas operações:

```
a + b
x = new y
```

- Tipos limitam as operações que podem ser feitas em um programa semanticamente válido

```
5 + true
```

- Tipos ajudam a construir programas confiáveis e eficientes

# Sistemas de Tipos

- Informalmente, um sistema de tipos consiste de:
  1. Um mecanismo para definir tipos e associá-los com certas contruções da linguagem
  2. Um conjunto de regras de equivalência, compatibilidade e inferência de tipos
- Toda construção com um valor associado deve ter um tipo

# Verificação de Tipos (*Type Checking*)

- Assegura que um programa obedece às regras de compatibilidade de tipos da linguagem
- Formas de verificação de tipos:
  - Tipagem forte x fraca
  - Tipagem estática x dinâmica
- Tipagem dinâmica oferece mais flexibilidade para o programador mas implica em um maior custo durante a execução do programa
  - Também adia a detecção de erros

# Definição de Tipos

- Algumas linguagens antigas de alto nível (e.g., Fortran 77, Algol 60 e Basic) não permitiam definir novos tipos
- Em Fortran, o tipo de uma variável é dado de acordo com o nome dela (e.g., *i* é uma variável inteira)
- Em algumas linguagens (e.g., Haskell, ML, Lisp, Smalltalk) o tipo de um valor é inferido automaticamente
- Na maioria das linguagens (e.g., C, Java, Pascal) é necessário declarar o tipo de um objeto

# Pensando sobre Tipos

- Podemos raciocinar sobre mais de um ponto de vista:
  - Visão denotacional: um tipo é simplesmente um conjunto de valores
  - Visão construtiva: um tipo é ou um dos tipos básicos pré-definidos (inteiros, caracteres, booleanos) ou um tipo composto criado através da aplicação de um construtor
  - Visão abstrata: um tipo é uma interface que consiste de uma série de operações com semântica bem definida.

# Questão de Implementação: Inteiros

- Como guardar pequenos intervalos inteiros?
- Em C e C++ há uma grande variedade de tipos inteiros:
  - char
  - short
  - int
  - long
  - long long
- Tamanho de um inteiro em C/C++ é dependente da implementação
- Em Java o intervalo de cada tipo inteiro é fixo

# Equivalência de Tipos

- Há duas formais principais de definir equivalência entre tipos:
  - Equivalência estrutural: baseada no conteúdo das definições de tipos
  - Equivalência por nome: cada definição (cada novo nome) introduz um novo tipo
- Equivalência por nome é a abordagem mais comum em linguagens modernas, como Java e C#
- A definição exata de equivalência estrutural depende da linguagem

# Equivalência Estrutural

```
type R1 = record a, b : integer end;

type R2 = record
  a, b : integer
end;

type R3 = record
  a : integer;
  b : integer;
end;
```

- *R1* deveria ser equivalente a *R2* e a *R3*?

# Equivalência Estrutural

- O tipo *str1* abaixo

```
type str1 = array [1..10] of char;
```

- deveria ser igual ao tipo *str2* a seguir?

```
type str2 = array [1..2*5] of char;
```

- Deveria também ser equivalent a *str3*?

```
type str3 = array [0..9] of char;
```

# Equivalência Estrutural

- A equivalência entre tipos é pensada de um modo orientado à implementação, um pouco baixo nível
- Dificuldade em distinguir tipos que o programador pensa que são diferentes mas que possuem a mesma estrutura

```
type student = record
  name, address : string
  age : integer

type school = record
  name, address : string
  age : integer

x : student;   y : school;
...
x := y
```

# Equivalência por Nome

- Se o programador escreveu duas definições de tipos, então esses tipos provavelmente devem ser diferentes
- Equivalência por nome e Aliasing
  - Os tipos abaixo são equivalentes?

```
TYPE new_type = old_type  (* Modula-2 *)
```

# Equivalência por Nome e Aliasing

```
TYPE stack_element = INTEGER;   (* alias *)
MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
    ...
PROCEDURE push(elem : stack_element);
    ...
PROCEDURE pop() : stack_element;
```

# Equivalência por Nome e Aliasing

```
TYPE celsius_temp = REAL;
     fahrenheit_temp = REAL;
VAR c celsius_temp;
    f : fahrenheit_temp;
...
f := c; (* isso provavelmente deveria ser um erro *)
```

# Equivalência por Nome e Aliasing

- Equivalência por nome estrita: tipos renomeados (*aliased types*) são considerados diferentes
- Equivalêcia por nome fraca: tipos renomeados são considerados equivalentes
- Ada permite os dois tipos de equivalência: Tipos derivados x Subtipo

```
subtype stack_element is integer;
...
type celsius_temp is new integer;
type fahrenheit_temp is new integer;
```

# Conversões de Tipos e *Casts*

- Em linguagens estaticamente tipadas, há contextos em que um valor de um certo tipo é esperado

```
a + b
foo(arg1, arg2, ..., argn)
```

- Para usar um valor de um tipo diferente do esperado pelo contexto, devemos explicitamente converter os tipos (fazer um *cast*)
  - Talvez código adicional precise ser executado

# Conversões de Tipos e *Casts*

- Conversão entre tipos
  1. Tipos estruturalmente equivalentes em uma linguagem que use equivalência por nome
  2. Tipos que possuem valores diferentes, mas os valores em comum são representados da mesma maneira
  3. Tipos possuem representação de baixo nível diferentes, mas apesar disso é possível definir uma correspondência entre eles

# Conversões de Tipos e *Casts*

```
n integer;          -- assuma 32 bits
r real;             -- assuma o padrão double da IEEE
type test_score = 0..100;
type celsius_temp is new integer;
c celsius_temp;

t := test_score(n); -- verificação em tempo de execução
n := integer(t);    -- nenhuma verificação
r := real(n) ;      -- conversão em tempo de execução
n := integer(r);    -- conversão em tempo de execução
n := integer(c);    -- nenhum código em tempo de execução
c := celsius_temp(n); -- nenhum código em tempo de execuçã
```

# Compatibilidade de Tipos

- Nem sempre a equivalência entre tipos é exigida, apenas a sua compatibilidade
  - Em uma atribuição, o tipo do lado direito deve ser compatível com o do lado esquerdo
  - Em uma chamada de função os tipos dos argumentos de uma função devem ser compatíveis com os tipos dos parâmetros
- A definição de compatibilidade varia muito de linguagem pra linguagem

# Compatibilidade de Tipos

- Em Ada um tipo $S$ é compatível com o tipo esperado $T$ se e somente se:
  1. $S$ e $T$ são equivalentes, ou
  2. um é subtipo do outro (ou ambos são subtipos do mesmo tipo base), ou
  3. ambos são arrays com os mesmos tipos e números de elementos em cada dimensão
- Em Pascal, uma regra extra de compatibilidade permite usar um inteiro onde um número real é esperado

# Coerção

- Conversão automática (implícita) de tipos

```
type weekday = (sun, mon, tue, wed, thu, fri, sat);
type workday = mon..fri;
d : weekday;
k : workday;
type calendar_column is new weekday;
c : calendar_column;
...
k := d; -- verificação em tempo de execução
d := k; -- nenhuma verificação
c := d; -- erro semântico estático
```

# Coerção

- Coerção é um tópico controverso
- Possibilidade de misturar tipos sem conversões explícitas enfraquece a segurança do sistema de tipos

```
short int s;
unsigned long int l;
char c;
float f;
double d;
...
s = l;
l = s;
s = c;
f = l;
d = f;
f = d;
```

# Sobrecarga e Coerção

- Um nome sobrecarregado por se referir a mais de objeto
- Ambiguidade deve ser resolvida pelo contexto
- O que acontece na expressão a seguir?

```
a + b
```

# Tipos de Referência Universal

- Várias linguagens possuem um tipo de referência universal
  - `(void *)` em C e C++
  - `Object` em Java
- Qualquer valor pode ser atribuído a um objeto do tipo referência universal
- O contrário deve ser feito com cuidado

```
Stack myStack =new Stack();
String s = "Hi Mom";
Foo f = new Foo();
...
myStack.push(s);
myStack.push(f);
...
s =(String) myStack.pop();
```

# Inferência de Tipos

- O que determina o tipo de uma expressão?
  - Respostas fáceis
  - Repostas não tão fáceis $\Rightarrow$ inferência de tipos
- Linguagens funcionais como ML, Miranda e Haskell possuem sistemas sofisticados de inferência de tipos

# Inferência de Tipos

- Sub-intervalos: qual o tipo de `a + b`?

```
type Atype = 0 .. 20;
     Btype = 10 .. 20;
var a : Atype;
    b : Btype;
```

- Inferência em C#:

```
var i = 123;                    //equiv. a  int i = 123;
var map = new Dictionary<int, string>();
//equiv. a  Dictionary<int, string> map= new Dic...
```

# Referências


- Programming Language Pragmatics (Michael Scott)
  - Capítulo 7