Matemática Discreta - 2009.2
Notas sobre Matematiquês vs. Pascal vs. C vs. Lua
Eduardo Ochs - PURO-UFF
Versão: 2009nov17, 23:30

Tipos

Em C tudo são números — até strings, arrays e funções são tratados como números internamente (são pointers, ou seja, números que indicam a posição na memória onde começa a representação daquele objeto). Cada variável tem um tipo fixo (inteiro, pointer para char, etc),

Em Pascal também, *internamente* — mas o sistema de tipos faz com que não possamos misturar inteiros, booleanos, funções, etc.

Em linguagens como Python, Ruby e Lua cada *objeto* tem um tipo fixo, mas cada variável é uma caixa que contem um objeto *de qualquer tipo*. Os resultados de expressões também são objetos: o resultado de `2+3` é um número, o resultado de `2==4` é um booleano (`true` ou `false`), e, no exemplo abaixo, o resultado de `m(2, 3)` um número (e o cálculo de `m(2, 3)` tem "efeitos colaterais"), o resultado de `m` é o objeto contido na variável `m` — uma função.

Matematiquês é mais parecido com linguagens como Lua do que com linguagens como C — em Matematiquês temos vários tipos diferentes de objetos (números, booleanos, listas, conjuntos, etc) e objetos de tipos diferentes são diferentes. Em C o "zero" e o "falso" são exatamente iguais!

Listas e conjuntos

Em linguagens de programação listas são fáceis de representar (arrays!) mas conjuntos são difíceis; às vezes o único modo de representar um conjunto numa linguagem de programação é fazer uma função que recebe um elemento e retorna ou "verdadeiro" ou "falso" — e interpretamos "verdadeiro" como "pertence" e "falso" como "não pertence".

Em Matematiquês nós sabemos manipular conjuntos — até conjuntos infinitos — e sabemos até comparar conjuntos: lembre que definimos $A = B$ como $A \subseteq B \wedge B \subseteq A$. Um programa que testasse se dois conjuntos são iguais testando se as suas "funções de pertencimento" são iguais levaria tempo infinito pra rodar.

Igualdade

O modo mais natural de testar se duas funções são iguais em Lua é ver se as duas são "a mesma" – isto é, se tanto uma quanto a outra estão no mesmo lugar da memória, e portanto correspondem exatamente às mesmas instruções. Duas funções com as mesmas instruções mas em lugares diferentes da memória — como a `f1` e a `f2` — são diferentes; seria possível inventar um outro tipo de função de comparação que respondesse que a `f1` e a `f2` são iguais, mas os criadores da linguagem resolveram definir o `==` do modo mais simples. Quanto às funções `f2` e `f3`, nós, humanos, rapidamente descobrimos que elas dão os mesmos resultados, mas como é que um computador poderia descobrir isso? É possível definir uma função que compara funções que reconhece que `f2` e `f3` são "iguais"?

Funções

Estamos vendo no curso de Matemática Discreta que em matematiquês funções são representadas como relações, e já vimos que relações são representadas como conjuntos; como sabemos comparar conjuntos sabemos quando duas funções são "iguais" em Matematiquês.

Até pouco tempo atrás no curso parecia que funções eram objetos de um tipo diferente dos outros, e que uma função nunca seria igual a um número, um valor de verdade, uma lista, ou um conjunto... Agora que nós vimos que funções "são" conjuntos nós descobrimos que, por exemplo, se f e g são funções definidas por:

$$\begin{array}{rrcl} f: & \{\,1,2,3\,\} & \to & \mathbb{R} \\ & x & \mapsto & 4x \\ g: & \{\,1,2,3\,\} & \to & \mathbb{Z} \\ & n & \mapsto & n \cdot 2^2 \end{array}$$

então:

$$f = g = \{\,(1,4),(2,8),(3,12)\,\}$$

Essa idéia de que "funções são conjuntos" é uma "decisão dos criadores da linguagem", como a decisão dos criadores do C de que "valores de verdade são números".

Note que é possível extrair o domínio e a imagem de f e g — $\{\,1,2,3\,\}$ e $\{\,4,8,12\,\}$ — a partir de $\{\,(1,4),(2,8),(3,12)\,\}$, mas não os contradomínios — $\mathbb{R}$ e $\mathbb{Z}$.

Erros, loops, efeitos colaterais

Em linguagens como Pascal e C, o domínio e o contradomínio são especificados na definição das funções, e o compilador cuida para que tanto os argumentos quanto o valor de retorno de uma função sejam sempre dos tipos certos, mas isso não garante que uma função retorna: ela pode entrar em loop e executar eternamente, ou ela pode dar um erro — por exemplo, `sqrt(2)` dá erro. Além disso funções podem ter "efeitos colaterais" quando executadas — como imprimir coisas e modificar variáveis — que fazem com seja diferente executá-las uma vez só ou duas...

Em matematiquês funções são representadas como conjuntos de pares que simplesmente dizem que valor a função retorna para cada valor que ela recebe. Aparentemente numa função definida por uma fórmula, como

$$\begin{array}{rrcl} h: & \mathbb{R} & \to & \mathbb{R} \\ & x & \mapsto & (((x+1)x+2)x+3)x+4 \end{array}$$

a fórmula é como um programinha que precisa ser executado a cada vez que precisamos do resultado — mas não é bem assim, é como se assim que definimos uma função f rodamos esse programinha (a fórmula) uma vez para cada valor do domínio, e aí construímos a representação da função como um conjunto de pares — que funciona como uma tabela — e a partir daí iremos sempre só consultar a tabela.

Problema (pra pensar sobre): em linguagens de programação a função `sqrt` leva '`real`'s em '`real`'s (ou '`float`'s em '`float`'s), e em matematiquês a função raiz quadrada vai de $[0, \infty)$ em $\mathbb{R}$, mas às vezes fingimos que ela vai de $\mathbb{R}$ em $\mathbb{R}$ e que ela dá "erro" para argumentos negativos... qual é o valor de $0 \cdot \sqrt{-2}$? Isso dá 0, erro, ou o quê?

Ordem de execução

Em linguagens de programação a ordem em que as instruções de um programa vão ser executadas é pré-definida, e é conhecida pelos programadores. Em matematiquês estamos vendo desde o início do curso que o valor de expressões matemáticas pode ser calculado de vários modos, em várias ordens, mas que o resultado é sempre igual... lembre dos nossos diagramas com '$\rightsquigarrow$'s, por exemplo:

$$\begin{array}{ccccc} (1+2)\cdot(3+4) & & \rightsquigarrow & & (1+2)\cdot 7 \\ \downarrow & & & & \downarrow \\ 3\cdot(3+4) & & \rightsquigarrow & & 3\cdot 7 \end{array} \qquad \begin{array}{c} \\ \\ \rightsquigarrow \\ \\ \rightsquigarrow \end{array} \qquad \begin{array}{c} \\ \\ 21 \\ \\ \end{array}$$

Redefinições

Em linguagens de programação variáveis podem ser redefinidas e mudarem de valor, e comandos como

```
contador = contador + 1;
```

são válidos; em matematiquês dentro de cada contexto assim que uma variável é definida fica impossível mudar o seu valor — o único modo de "mudar o valor" dela é fechar o contexto onde ela foi definida.

Expressões como

$$\begin{array}{c} \{\, x \in A \mid P(x) \,\} \\ \forall x \in A.\, P(x) \\ \exists x \in A.\, P(x) \\ x \mapsto 4x \qquad \text{(numa definição de função)} \end{array}$$

criam contextos, e algumas expressões em Português como "Seja $x = ...$; então..." também criam.

Note que estes "contextos" funcionam um pouco como os "`begin ... end`"s e as declarações de procedures e funções do Pascal: quando damos nomes para os argumentos para procedures e funções, ou quando criamos variáveis locais, estamos criando variáveis cujos nomes "deixam de existir" quando o contexto é fechado.

Strings

Linguagens de programação têm strings, que se comportam como arrays de caracteres; podemos definir strings em matematiquês mesmo sem introduzirmos nenhum tipo novo (da mesma forma que relações são definidas como conjuntos) definindo strings como listas de números — por exemplo, "`Hello`" poderia ser

só uma representação conveniente para a lista (72, 101, 108, 108, 111) — mas infelizmente poucos livros de Matemática Discreta fazem isso... o Scheinerman não faz, então não tratamos de strings no curso.

Conjuntos infinitos

Em matematiquês podemos ter listas e conjuntos infinitos — $\mathbb{R}$, $\mathbb{N}$ e $\mathbb{Z}$ são infinitos, por exemplo — mas em linguagens de programação não — nem todo real pode ser representado como `real` ou `float`, nem todo inteiro pode ser representado como `integer`. (Pense: como é que eles seriam representados na memória?)

Em linguagens como Lua podemos ter um array "circular", `A`, no qual `A[1] = A`; em matematiquês um conjunto A não pode ser membro de si mesmo (conjuntos "circulares" causavam problemas lógicos, então "consertaram" as regras que diziam como conjuntos podem ser construídos para que elas não permitissem mais que construíssemos conjuntos que contivessem a si mesmos... mas não vamos poder ver estas regras com detalhes neste curso).

```
(* Programa em Pascal *)

program teste;
function m(a: integer; b: integer): integer;
  begin WriteLn("  ", a, "*", b, " -> ", a*b); return(a*b); end;
function s(a: integer; b: integer): integer;
  begin WriteLn("  ", a, "+", b, " -> ", a+b); return(a+b); end;
function P1(a: integer): boolean;
  begin WriteLn("  ", a, "*", a, " < 10 -> ", a*a<10); return(a*a<10); end;
function P2(a: integer): boolean;
  begin WriteLn("  ", a, "=4 -> ", a=4); return a=4; end;
function P3(a: integer): boolean;
  begin WriteLn("  ", a, ">=9 -> ", a>=9); return a>=9; end;
procedure testaordem;
  begin WriteLn("2*3 + 4*5:");
        WriteLn(" -> ", s(m(2, 3), m(4, 5)));
        WriteLn;
  end;
procedure f123(function f(a: integer; b: integer): integer);
  begin WriteLn("f(f(1,2), f(3,4)):");
        WriteLn(" -> ", f(f(1, 2), f(3, 4)));
        WriteLn;
  end;
procedure existe_a_em_A_tal_que(function P(a: integer): boolean);
  begin WriteLn("Existe a em {1, 2, 4} tal que P:");
        WriteLn("-> P(1) or P(2) or P(4)");
        WriteLn(" -> ", P(1) or P(2) or P(4));
        WriteLn;
  end;

begin
  testaordem;
  f123(s);
  f123(m);
  existe_a_em_A_tal_que(P1);
  existe_a_em_A_tal_que(P2);
  existe_a_em_A_tal_que(P3);
end.
```

```
(* Output do programa em Pascal:

2*3 + 4*5:
 ->   4*5 -> 20
  2*3 -> 6
  6+20 -> 26
26

f(f(1,2), f(3,4)):
 ->   3+4 -> 7
  1+2 -> 3
  3+7 -> 10
10

f(f(1,2), f(3,4)):
 ->   3*4 -> 12
  1*2 -> 2
  2*12 -> 24
24

Existe a em {1, 2, 4} tal que P:
-> P(1) or P(2) or P(4)
  1*1 < 10 -> True
 -> True

Existe a em {1, 2, 4} tal que P:
-> P(1) or P(2) or P(4)
  1=4 -> False
  2=4 -> False
  4=4 -> True
 -> True

Existe a em {1, 2, 4} tal que P:
-> P(1) or P(2) or P(4)
  1>=9 -> False
  2>=9 -> False
  4>=9 -> False
 -> False

*)
```

```
/* Programa em C: */

#include <stdio.h>
int m(int a, int b) { printf("  %d*%d -> %d\n", a, b, a*b); return a*b; }
int s(int a, int b) { printf("  %d+%d -> %d\n", a, b, a+b); return a+b; }
int P1(int a) { printf("  %d*%d < 10 -> %d\n", a, a, a*a<10); return a*a<10; }
int P2(int a) { printf("  %d==4 -> %d\n", a, a==4); return a==4; }
int P3(int a) { printf("  %d>=9 -> %d\n", a, a>=9); return a>=9; }
void testaordem(void) {
  printf("2*3 + 4*5: \n");
  printf(" -> %d\n\n", s(m(2, 3), m(4, 5)));
}
void f123(int(*f)()) {
  printf("f(f(1,2), f(3,4)):\n");
  printf(" -> %d\n\n", f(f(1, 2), f(3, 4)));
}
void existe_a_em_A_tal_que(int(*P)()) {
  printf("Existe a em {1, 2, 4} tal que P:\n");
  printf("-> P(1) || P(2) || P(4)\n");
  printf(" -> %d\n\n", P(1) || P(2) || P(4));
}

main() {
  testaordem();
  f123(s);
  f123(m);
  existe_a_em_A_tal_que(P1);
  existe_a_em_A_tal_que(P2);
  existe_a_em_A_tal_que(P3);
}
```

```
/* Output do programa em C:

2*3 + 4*5:
  4*5 -> 20
  2*3 -> 6
  6+20 -> 26
 -> 26

f(f(1,2), f(3,4)):
  3+4 -> 7
  1+2 -> 3
  3+7 -> 10
 -> 10

f(f(1,2), f(3,4)):
  3*4 -> 12
  1*2 -> 2
  2*12 -> 24
 -> 24

Existe a em {1, 2, 4} tal que P:
-> P(1) || P(2) || P(4)
  1*1 < 10 -> 1
 -> 1

Existe a em {1, 2, 4} tal que P:
-> P(1) || P(2) || P(4)
  1==4 -> 0
  2==4 -> 0
  4==4 -> 1
 -> 1

Existe a em {1, 2, 4} tal que P:
-> P(1) || P(2) || P(4)
  1>=9 -> 0
  2>=9 -> 0
  4>=9 -> 0
 -> 0

*/
```

```
-- Programa em Lua:

function printf(...) write(format(...)) end
function str(obj) return tostring(obj) end

function m(a, b) printf("  %d*%d -> %d\n", a, b, a*b); return a*b end
function s(a, b) printf("  %d+%d -> %d\n", a, b, a+b); return a*b end
function P1(a) printf("  %d*%d < 10 -> %s\n", a, a, str(a*a<10)); return a*a<10 end
function P2(a) printf("  %d==4 -> %s\n", a, str(a==4)); return a==4 end
function P3(a) printf("  %d>=9 -> %s\n", a, str(a>=9)); return a>=9 end
function testaordem()
    printf("2*3 + 4*5: \n")
    printf(" -> %d\n\n", s(m(2, 3), m(4, 5)))
  end
function f123(f)
    printf("f(f(1,2), f(3,4)):\n")
    printf(" -> %d\n\n", f(f(1, 2), f(3, 4)))
  end
function existe_a_em_A_tal_que(P)
    printf("Existe a em {1, 2, 4} tal que P:\n")
    printf("-> P(1) || P(2) || P(4)\n")
    printf(" -> %s\n\n", str(P(1) or P(2) or P(4)))
  end

testaordem()
f123(s)
f123(m)
existe_a_em_A_tal_que(P1)
existe_a_em_A_tal_que(P2)
existe_a_em_A_tal_que(P3)

function f1(a) return 4*a end
function f2(a) return 4*a end
function f3(a) return a*4 end
print(f1, f2, f1==f2)
print(f1==f1)
```

Log do programa em Lua (rodado interativamente):

```
Lua 5.1.2  Copyright (C) 1994-2007 Lua.org, PUC-Rio
> -- Programa em Lua:
>
> function printf(...) write(format(...)) end
> function str(obj) return tostring(obj) end
>
> function m(a, b) printf("  %d*%d -> %d\n", a, b, a*b); return a*b end
> function s(a, b) printf("  %d+%d -> %d\n", a, b, a+b); return a*b end
> function P1(a) printf("  %d*%d < 10 -> %s\n", a, a, str(a*a<10)); return a*a<10 end
> function P2(a) printf("  %d==4 -> %s\n", a, str(a==4)); return a==4 end
> function P3(a) printf("  %d>=9 -> %s\n", a, str(a>=9)); return a>=9 end
> function testaordem()
>>     printf("2*3 + 4*5: \n")
>>     printf(" -> %d\n\n", s(m(2, 3), m(4, 5)))
>>   end
> function f123(f)
>>     printf("f(f(1,2), f(3,4)):\n")
>>     printf(" -> %d\n\n", f(f(1, 2), f(3, 4)))
>>   end
> function existe_a_em_A_tal_que(P)
>>     printf("Existe a em {1, 2, 4} tal que P:\n")
>>     printf("-> P(1) || P(2) || P(4)\n")
>>     printf(" -> %s\n\n", str(P(1) or P(2) or P(4)))
>>   end
>
> testaordem()
2*3 + 4*5:
  2*3 -> 6
  4*5 -> 20
  6+20 -> 26
 -> 120

> f123(s)
f(f(1,2), f(3,4)):
  1+2 -> 3
  3+4 -> 7
  2+12 -> 14
 -> 24

> f123(m)
f(f(1,2), f(3,4)):
  1*2 -> 2
  3*4 -> 12
  2*12 -> 24
 -> 24
```

```
> existe_a_em_A_tal_que(P1)
Existe a em {1, 2, 4} tal que P:
-> P(1) || P(2) || P(4)
  1*1 < 10 -> true
 -> true

> existe_a_em_A_tal_que(P2)
Existe a em {1, 2, 4} tal que P:
-> P(1) || P(2) || P(4)
  1==4 -> false
  2==4 -> false
  4==4 -> true
 -> true

> existe_a_em_A_tal_que(P3)
Existe a em {1, 2, 4} tal que P:
-> P(1) || P(2) || P(4)
  1>=9 -> false
  2>=9 -> false
  4>=9 -> false
 -> false

>
> function f1(a) return 4*a end
> function f2(a) return 4*a end
> function f3(a) return a*4 end
> print(f1, f2, f1==f2)
function: 0x851cba8 function: 0x8516f90 false
> print(f1==f1)
true
>
```