Bem-vindos

Linguagens de Programação

1 de março de 2019

Tópicos

Objetivo

Aprender conceitos fundamentais comuns a diversas LP

- Vamos usar várias linguagens de programação para exercitar tais conceitos
- ... para perceber como representam tais conceitos
- ... para ajudar que vocês sejam melhores desenvolvedores em qualquer LP

Desenvolvimento do Curso

- Estrutura do Curso: (Prof. Dan Grossman, University of Washington)
- Apostila: vamos utilizar a apostila preparada pelo Prof. D. Grossman (com a sua permissão)
- Serão acrescentados alguns conteúdos complemetares
- Indispensável: por a mão na massa e fazer todos os exercícios de programação
- A maior parte do curso será sobre linguagens funcionais (ex, ML e Racket)
- Também falaremos algo sobre OOP (Ruby)
- Os conceitos de interesse serão apresentados junto com a LP

Atenção

Muita Atenção

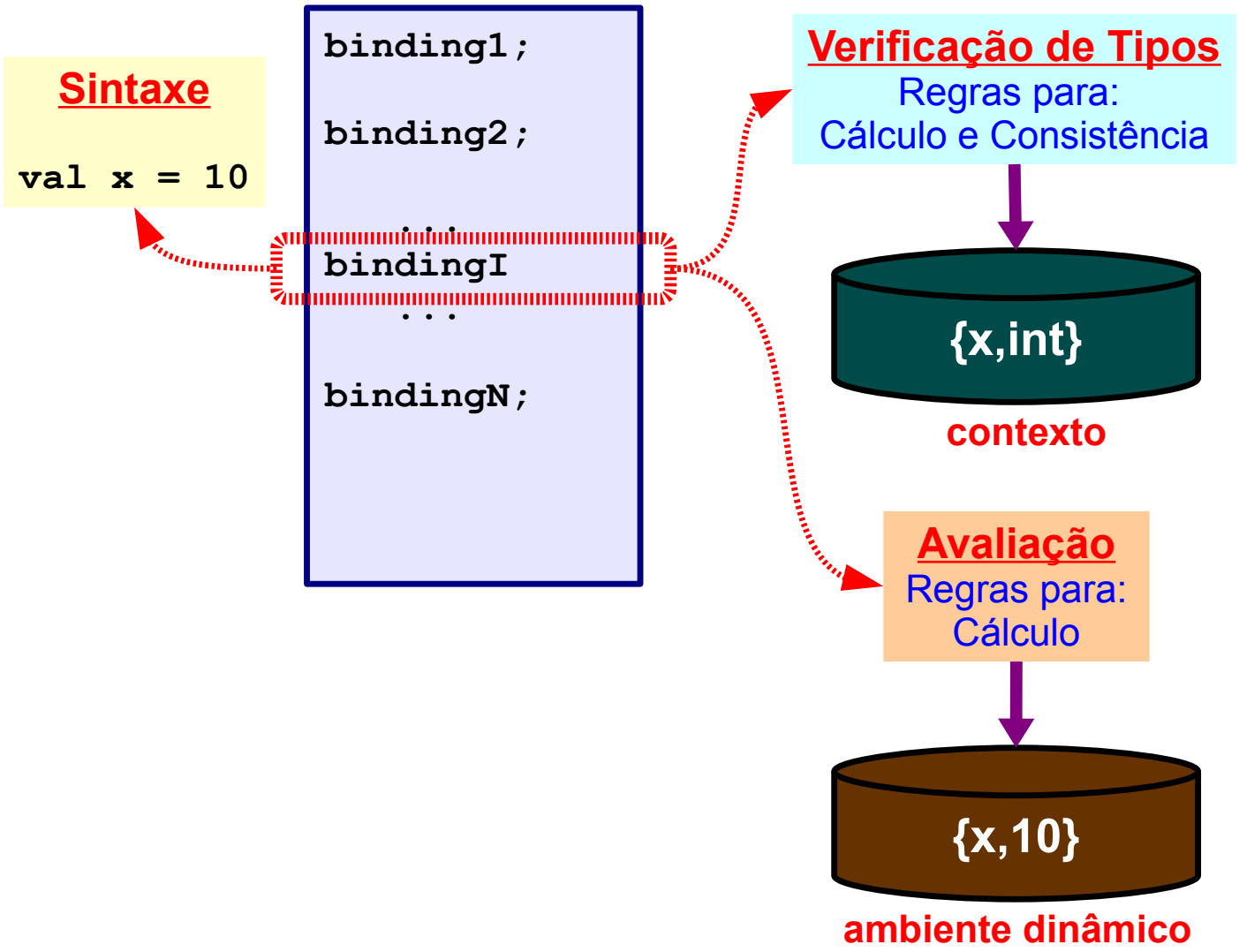
- Nos conceitos apresentados e nas palavras que os denominam
- Evite (por enquanto) relacionar os novos conceitos com linguagens de programação já conhecidas

Expressões e Bindings

Um programa é uma sequência de bindings (associação)

- binding não é atribuição!!!
- cada binding é type checked e avaliado
- ambiente estático (static environment, contexto): usado para verificação do tipo
  - basicamente, tipos das associações precedentes
- ambiente dinâmico (dynamic environment): usado para a avaliação
  - basicamente, os valores das associações precedentes

Significado do Programa



Associação à Variável (variable binding)

val `x` : *tipo* = *e*;

- Sintaxe:
  - `val` é palavra reservada
  - tipo é uma declaração de tipo (próximos capítulos)
  - `x` é um identificador (regras para identificador) *e* é expressão (regras para expressão)
- Semântica (quais as regras para?):
  - Verificação de tipos:
    - Cálculo do tipo de *e*: tipo *t* (Pode falhar)
    - Verifica se *t* é "compatível" com *tipo*
    - SE = SE $\bigcup$ {x possui o tipo t}
  - Avaliação:
    - o valor de *e* é *v*
    - DE = DE $\bigcup$ {*x* possui o valor *v*}

Expressões

Constante inteira

- Sintaxe: sequência de algarismos numéricos (ex: 10)
- Verificação de tipos: o tipo é int
- Avaliação: o próprio valor

Constantes lógicas

- Sintaxe: true , false
- Verificação de tipos: o tipo é bool
- Avaliação: o próprio valor

Expressões

Adição

- Sintaxe: $e1 + e2$
- Verificação de tipos:
  - e1 e e2 devem ser do tipo int, cc ERRO
  - tipo da expressão: int
- Avaliação:
  - $v1 = valor(e1)$
  - $v2 = valor(e2)$
  - valor da adição: $v1 + v2$

Variável

- Sintaxe: identificador (regras para identificador)
- Verificação de tipos: o tipo variável no contexto corrente
- Avaliação: o valor da variável no ambiente dinâmico corrente

Expressões

Condicional

- Sintaxe: if e1 then e2 else e3
- Verificação de tipos:
  - tipo(e1)=bool
  - tipo(e2)=tipo(e3)=t, cc ERRO
  - tipo da expressão: t
- Avaliação:
  - $v1 = valor(e1)$
  - se v1 então valor(e2) senão valor(e3)

Expressões

Comparação ($<$)

- Sintaxe: $e1 < e2$
- Verificação de tipos:
  - e1 e e2 devem ser do mesmo tipo t, cc ERRO
  - tipo t deve permitir a comparação
  - tipo da expressão: bool
- Avaliação:
  - $v1 = valor(e1)$
  - $v2 = valor(e2)$
  - valor da comparação: $v1 < v2$

Variáveis

Variáveis são Imutáveis

- *val* $x = 8 + 9$
- ...x "mapeia" a 17
- *val* $x = 19$ (*shadows*)
- ...novo binding, cria novo ambiente

Associação de função

```
fun x0 (x1 :  t1, ...  , xn:  tn) = e
```

```
fun pow (x:int, y:int) =
      if y=0
          then 1
          else x*pow(x,y-1)
(* correto para y >= 0 *)
```

Verificação de Tipo

- faz verificação do tipo de e no contexto: $SE \bigcup \{(x1, t1), \ldots, (xn, tn), (x0, t1 * \ldots * tn \rightarrow t\}$
- t?: inferência de tipos. (Aguardem próximos capítulos)!
- tipo(e) deve ser t
- associação {x0,(x1,t1),. . . ,(xn,tn), (x0,t1*. . . *tn→t} é acrescentado ao SE

Avaliação

A função é um valor!

Chamada de função

`e0 (e1,...,en)`

Parêntesis são opcionais caso exista apenas 1 argumento

Verificação de tipo

- t1=tipo(e1), ..., tn=tipo(en)
- tipo(e0) deve ser da forma $t1 * ... * tn \rightarrow t$
- tipo do resultado da função: t

Avaliação

- Usa-se o DE no ponto da chamada para avaliar e0,...,en
- valor(v0) deve ser função (supondo verificação de tipo ok)
- corpo da função é avaliado

Exemplo

Correto para y não-negativo

```
fun pow (x:int, y:int) =
      if y=0
          then 1
          else x*pow(x,y-1)
```

```
fun cube (x: int) = pow(x,3)
val rsp = cube(4)
```

Pares e Outras Tuplas

`(e1,e2)`

- tipo: `t1 * t2`
- valor: `(v1,v2)`

Recuperando partes

- #1(e1,e2) = v1; tipo é t1
- #2(e1,e2) = v2; tipo é t2

Exemplos

Resultado de função pode ser uma tupla

```
fun swap(pr: int * bool) =
      (#2 pr, #1 pr)

fun sum_two_pairs (pr1: int*int, pr2: int*int) =
      (#1 pr1) + (#2 pr1) + (#1 pr2) + (#2 pr2)

fun div_mod(x: int, y: int) =
       (x div y, x mod y)
```

Tuplas podem ter vários elementos

- #4 (“ze”,21,76.5,true,“uel”)

Listas

Características

- Sequência de elementos do mesmo tipo
  - tipo do elemento: `'a`
  - tipo da lista: `'a list`
- lista vazia: `[ ]`
- lista não vazia: [e1,e2,...,eN]

Funções básicas

- `null lst`: true se lst estiver vazia
- `hd lst`: retorna primeiro elemento da lista. Exceção se vazia
- `tl lst`: retorna lista sem o primeiro elemento
- `el::lst`: retorna lista acrescentando o elemento el ao inicio da lista lst

Exemplos

Funções que processam ou retornam listas

```
fun sum_list (xs : int list) =
    if null xs
    then 0
    else hd(xs) + sum_list(tl xs)

fun countdown (x : int) =
    if x=0
    then [ ]
    else x::countdown(x-1)

fun append (xs : int list, ys : int list)=
    if null xs
    then ys
    else (hd xs) :: append(tl xs, ys)
```

Escopo Local

`let b1 b2 ... bn in e end`

- Define associações locais
- Escopo: bindings subsequentes, e
- São expressões (podem ser sub-expressões)

Exemplo

```
fun area_retangulo (x1: int, y1: int, x2: int, y2: int) =
let
    val dx = x2 - x1
    val dy = y2 - y1
in
    dx * dy
end
```

Options

Retorna o maior elemento da lista

```
fun max(xs : int list) =
    if null xs
    then O que retornar?
    else ...
```

max [ ]

- 0?
- -1?

  ?
- ...

Options: SOME, NONE

Exemplo: int option

```
fun better_max(xs : int list) =
    if null xs
    then NONE
    else
        let val tl_ans = better_max(tl xs)
        in
            if isSome tl_ans andalso valOf tl_ans > hd xs
            then tl_ans
            else SOME (hd xs)
        end
```

"Mutação"

- Não se pode mudar o conteúdo de uma associação, tupla ou lista
- Novo binding por "sobrepor-se" ao anterior
  - Não afeta o código que usa o binding "antigo"

Benefícios da ausência de mutação

- Compartilhamento e *aliasing* se tornam irrelevantes
- (Vc já ouviu: Construtor de Cópia? Cópia rasa? Cópia profunda?)

Exercício:

- O que são "aliases"?
- Por que C++ possui construtor de cópia?
- Qual a relação entre "alias" e "cópias rasa e profunda"?