

Aula 3: Tipos de Dados em OCaml

UC: Programação Funcional

2023-2024

Tipos de dados compostos

- Até agora vimos: inteiros, Booleanos, variáveis, condicionais e funções
 - Agora vamos ver: formas de construir dados com múltiplas partes
- ➡
- **Tuplos**: número fixo de “peças” com tipos possivelmente diferentes
 - **Listas**: qualquer número de “peças”, todas com o mesmo tipo
- Mais à frente: formas mais gerais de criar dados compostos (árvores)

Pares (2-tuplo)

Sintaxe: $(e1, e2)$ onde $e1$ e $e2$ são expressões

Verificação do Tipo

- Se $e1$ tem tipo $t1$ e $e2$ tem tipo $t2$
- Então $(e1, e2)$ tem tipo $t1 * t2$
- Senão, reportar erro e falhar (acontece apenas se $e1$ ou $e2$ não verificarem o tipo)

Avaliação

- Se $e1$ avaliar no valor $v1$ e $e2$ avaliar no valor $v2$
- Então $(e1, e2)$ avalia para $(v1, v2)$

Estrela

- * é um construtor de tipos : constrói tipos de tuplos a partir de outros tipos
 - Portanto `int * bool`, `string * int`, `int * (int * int)`, são 3 do número infinito de tipos que podemos construir com *
- Os tipos dos tuplos também são conhecidos como tipos de produtos
- Mas a relação com a multiplicação é obscura e é melhor ignorá-la
 - Basta utilizar o mesmo carácter numa sintaxe não relacionada
 - `3*4` é uma expressão que avalia em 12, `int*int` é um tipo

Pares Aninhados

Os pares podem ser aninhados

- O aninhamento **Não é** uma nova funcionalidade!
- Simplesmente uma consequência das regras de sintaxe e semântica que estabelecemos

`((1,4),(2,(true,3)) : (int * int) * (int * (bool * int)))`

- A Composição é uma característica distintiva de um bom desenho
 - Frequentemente da ortogonalidade

Pares

Sintaxe: `fst e` e `snd e`

- Onde `e` é uma expressão

Verificação do Tipo

- Se `e` tem tipo `t1 * t2`, então
- `fst e` tem tipo `t1`
- `snd e` tem tipo `t2`

Avaliação

- Se `e` avalia para um par de valores `(v1, v2)`, então
- `fst e` avalia para `v1`
- `snd e` avalia para `v2`

Nota: `fst` e `snd` são na verdade funções de uma biblioteca, mas vamos considerar que são “built in” na linguagem por mais umas semanas.

Tuplos

Sintaxe: (e_1, \dots, e_N)

- Onde e_1, \dots, e_N são expressões e $N \geq 2$
- Isto é uma nova funcionalidade: $(5, \text{true}, 7)$ é um 3-tuplo (triplo), e não qualquer tipo de par aninhado

Verificação do Tipo

- Se e_1 tem tipo t_1 e ... e e_N tem tipo t_N
- Então (e_1, \dots, e_N) tem tipo $t_1 * \dots * t_N$
- Senão, reportar erro e falhar

Avaliação

- Se e_1 avalia para o valor v_1 e ... e e_N avalia para o valor v_N
- Então (e_1, \dots, e_N) avalia para (v_1, \dots, v_N)


Pares Aninhados vs. Tuplos

- OCaml não precisava de tuplos, mas eles podem ser convenientes.
- Os parênteses são importantes em tipos de tuplos e expressões de tuplos!

```
let x = (5,7,9)
(* x : int*int*int *)
let seven = snd3 x
(* snd x não tipa *)
```

```
let x = (5,(7,9))
(* x : int*(int*int) *)
let seven = fst (snd x)
let pr     = snd x
(* snd3 x não tipa *)
```


Tipos de dados compostos

- Até agora vimos: inteiros, Booleanos, variáveis, condicionais e funções
- Agora vamos ver: formas de construir dados com múltiplas partes
 - **Tuplos**: número fixo de “peças” com tipos possivelmente diferentes
 -  ◦ **Listas**: qualquer número de “peças”, todas com o mesmo tipo
- Mais à frente: formas mais gerais de criar dados compostos (árvores)

Listas

Elementos separados por ponto e vírgula “ ; ” ⚠
mensagens de erro muito estranhas, caso
contrário ⚠

Sintaxe: [e_1 ; ...; e_N]

- Onde e_1 , ..., e_N são expressões

Verificação do Tipo

- Se e_1 tem tipo t e ... e e_N tem tipo t
- Então [e_1 ; ...; e_N] tem tipo t **list**
- Senão, reportar erro e falhar

Todos do mesmo tipo t

Avaliação

- Se e_1 avalia no valor v_1 e ... e e_N avalia no valor v_N
- Então [e_1 ; ...; e_N] avalia em [v_1 ; ...; v_N]

Listas de valores são valores

Listas

Sintaxe: `[e1; ...; eN]`

- Onde `e1`, `...`, `eN` são expressões

Verificação do Tipo

- Se `e1` tem tipo `t` e ... e `eN` tem tipo `t`
- Então `[e1; ...; eN]` tem tipo `t list`
- Senão, reportar erro e falhar

Avaliação

- Se `e1` avalia no valor `v1` e ...
- Então `[e1; ...; eN]` avalia a

Novo tipo!

`t list`

Outro construtor de tipo (`list`) para construir o tipo de listas cujos elementos têm o tipo `t`.

Exemplos:

```
bool list    int list
(int * (int * int)) list
int list list
```

Listas

N pode ser zero!
[] é a lista vazia para
qualquer tipo.

Sintaxe: [e1; ...; eN]

- Onde e1, ..., eN são expressões

Verificação do Tipo

- Se e1 tem tipo t e ... e eN tem tipo t
- Então [e1; ...; eN] tem tipo t list
- Senão, reportar erro e falhar

Avaliação

- Se e1 avalia no valor v1 e ... e eN avalia no valor vN
- Então [e1; ...; eN] avalia em [v1; ...; vN]

Lista Vazia []

- Se e_1 tem tipo t e ... e e_N tem tipo t
- Então $[e_1; \dots; e_N]$ tem tipo t list

Portanto se **todos os elementos de [] têm tipo t** , então [] tem tipo t list

- [] tem tipo `int list`, `bool list`, `int list list`, `(bool*int) list`, ...
- Em OCaml “qualquer tipo de lista” escreve-se `'a list` (ou `'b list`, `'c list`, ...)
 - Mas todos os elementos de qualquer lista têm o mesmo tipo
- Veremos muito estes tipos polimórficos em OCaml

Construir Listas com o construtor (::)

Sintaxe: $e1 :: e2$

- Onde $e1$ e $e2$ são expressões

Verificação do Tipo

- Se $e1$ tem tipo t e $e2$ tem tipo t list
- Então $e1 :: e2$ tem tipo t list
- Senão, reportar erro e falhar

Avaliação

- Se $e1$ avalia em $v1$ e $e2$ avalia em $[v2; \dots; vN]$
- Então $e1 :: e2$ avalia em $[v1; v2; \dots; vN]$

“Criar (construir) uma lista com mais um elemento à frente”

Duas Formas de Construir Listas?

- `::` é mais comum em código
- `::` e `[]` é tudo o que necessitamos
 - `e1::e2::e3::...::en::[]` é o mesmo que `[e1;e2;e3;...;en]`
 - Primeiro de muitos exemplos de “açúcar sintático”
- Lista de valores é impressa por `[v1;v2;v3;...;vn]`

Listas

- Para já, vamos *utilizar* listas com estas características.
 - Testar se uma lista está vazia com `e = []`
 - Obter o primeiro elemento de uma *lista não vazia* com a função `List.hd`
 - Obter o resto da *lista não vazia* com a função `List.tl`
 - (uma lista com tudo o que se segue ao primeiro elemento)
- `List.hd` e `List.tl` levantará uma exceção se receber uma lista vazia
- Mais tarde, usaremos o *pattern-matching* para fazer o mesmo de forma mais segura e elegante

Teste de Listas Vazias

Sintaxe: $e = []$

- Onde e é uma expressão

Verificação do Tipo

- Se e tem tipo t list
- Então $e = []$ tem tipo bool
- Senão reportar erro e falhar

Avaliação

- Se e avaliar em [], então $e = []$ avalia em true
- Senão $e = []$ avalia em false

Obter o primeiro elemento da Lista

Sintaxe: `List.hd e`

- Onde `e` é uma expressão

Verificação do Tipo

- Se `e` tem tipo `t list`
- Então `List.hd e` tem tipo `t`
- Senão, reportar erro e falhar

Definição alternativa:

`List.hd` tem tipo `'a list -> 'a`

Avaliação

- Se `e` avaliar no valor `[v1; v2; ...; vN]`
- Então `List.hd e` avalia no valor `v1`
- Senão `e` avalia em `[]`, portanto `List.hd e` lança uma exceção

Obter os restantes elementos da Lista

Sintaxe: `List.tl e`

- Onde `e` é uma expressão

Verificação do Tipo

- Se `e` tem tipo `t list`
- Então `List.tl e` tem tipo `t list`
- Senão, reportar erro e falhar

Definição alternativa:

`List.tl` has type `'a list -> 'a list`

Avaliação

- Se `e` avalia no valor `[v1; v2; ...; vN]`
- Então `List.tl e` avalia no valor `[v2; ...; vN]`
- Senão `e` avalia em `[]`, portanto `List.tl e` lança uma exceção

Recursividade Novamente!

- As funções sobre listas são normalmente recursivas
 - A única forma de “chegar a todos os elementos”
- Receita: responder a duas perguntas
 - Qual deve ser a resposta para uma lista vazia?
 - Muitas vezes, pensar no tipo de retorno dá uma boa pista (caso base)
 - Qual deve ser a resposta para uma lista não vazia?
 - Tipicamente, isto será em termos da resposta para a recursividade terminal (recursão)

Registos (Records)

```
type int_pair = {first : int; second : int}
let sum_int_pr x = x.first + x.second
let pr1 = {first = 3; second = 4}
let _ = sum_int_pr pr1 + sum_int_pr {first=5;second=6}
```

Um construtor de tipos para dados/código polimórficos:

```
type 'a pair = {a_first : 'a; a_second : 'a}
let sum_pr f x = f x.a_first + f x.a_second
let pr2 = {a_first = 3; a_second = 4} (* par de inteiros *)
let _ = sum_int_pr pr1 + sum_pr (fun x->x) {a_first=5;a_second=6}
```

Código Polimórfico

```
type 'a pair = {a_first : 'a; a_second : 'a}
let sum_pr f x = f x.first + f x.second
let pr2 = {a_first = 3; a_second = 4}
let pr3 = {a_first = "hi"; a_second = "mom"}
let pr4 = {a_first = pr2; a_second = pr2}
let sum_int = sum_pr (fun x -> x)
let sum_str = sum_pr String.length
let sum_int_pair = sum_pr sum_int
let _ = print_i_nl (sum_int pr2)
let _ = print_i_nl (sum_str pr3)
let _ = print_i_nl (sum_int_pair pr4)
```

Tipos de dados

- Os registos constroem novos tipos através de “cada um” dos tipos existentes
- Também são necessários novos tipos através de “um dos” tipos existentes
 - *enums* ou *unions* (com etiquetas) em C
- Em OCaml fazemos isso diretamente; as etiquetas são os *constructores*
 - Um tipo é chamado de *tipo de dados* (datatype)

```
type food = Foo of int | Bar of int_pair
           | Baz of int * int | Quux

let foo3      = Foo (1 + 2)
let bar12     = Bar pr1
let baz1_120  = Baz (1, fact 5)
let quux      = Quux (* não é muito útil *)

let is_a_foo x =
  match x with (* muito útil *)
    Foo i      -> true
  | Bar pr     -> false
  | Baz (i, j) -> false
  | Quux       -> false
```

Tipos de dados

- Nota de sintaxe: Construtores iniciam em maiúsculas, variáveis não
- Utilize o construtor para criar um valor do tipo
- Utilizar o “pattern-matching” para usar um valor do tipo
 - A única forma de o fazer
 - O “pattern-matching” é, de facto, muito mais poderoso

Booleanos

Tipo de dados predefinido (infringe as regras de capitalização):

```
type bool = true | false
```

`if` é apenas açúcar sintático para `match` (mas com melhor estilo):

- `if e1 then e2 else e3`
- `match e1 with`
 - `true -> e2`
 - `| false -> e3`

Tipos Recursivos

Um tipo de dados pode ser recursivo, permitindo estruturas de dados de tamanho ilimitado

E pode ser polimórfico, tal como os registos

```
type int_tree = Leaf
                | Node of int * int_tree * int_tree
type 'a lst = Null
            | Cons of 'a * 'a lst

let lst1 = Cons(3, Null)
let lst2 = Cons(1, Cons(2, lst1))
(* let lst_bad = Cons("hi", lst2) *)
let lst3 = Cons("hi", Cons("mom", Null))
let lst4 = Cons (Cons (3, Null),
                  Cons (Cons (4, Null), Null))
```

Funções Recursivas (tamanho)

```
type 'a lst = Null
           | Cons of 'a * 'a lst

let rec length lst = (* 'a lst -> int *)
  match lst with
  | Null -> 0
  | Cons(x,rest) -> 1 + length rest
```

Funções Recursivas (soma)

```
type 'a lst = Null
           | Cons of 'a * 'a lst

let rec sum lst = (* int lst -> int *)
  match lst with
  | Null -> 0
  | Cons(x,rest) -> x + sum rest
```

Funções Recursivas (acrescentar elementos)

```
type 'a lst = Null
            | Cons of 'a * 'a lst

let rec append lst1 lst2 =
  (* 'a lst -> 'a lst -> 'a lst *)
  match lst1 with
  | Null -> lst2
  | Cons(x, rest) -> Cons(x, append rest lst2)
```

Construções Incorporadas (built-in)

Na verdade o tipo `'a list` está incorporado na linguagem:

- Null é escrito por `[]`
- **Cons** (**x**, **y**) é escrito por `x :: y`
- E açúcar sintático para uma lista de literais `[5; 6; 7]`

```
let rec append lst1 lst2 = (* incorporado na linguagem, infixo @ *)
  match lst1 with
  | [] -> lst2
  | x::rest -> x :: append rest lst2
```

Até agora

- Temos praticamente tudo o que precisamos
 - Funções de ordem superior
 - Records
 - Tipos de dados recursivos
- Alguns pormenores importantes para a próxima aula
 - Tuplos vs Records
 - Padrões (pattern-matching)
 - Exceções
- Depois, módulos (simples) ...

Créditos para Dan Grossman.