

Aula 5: Variantes e Correspondência de Padrões em OCaml

UC: Programação Funcional

2023-2024

Até agora vimos

- Já sabemos como construir vários tipos

Tipos base: `int` `bool` `float` `string`

Construtores de Tipos: `t1 -> t2` `t1 * t2 * ... * tn`

`t list` `t option`

- Algumas expressões

`34` `true` `(e1,e2,...,en)` `[]` `e1::e2` `x`

`f(e1,e2,...,en)` `e1+e2` `if e1 then e2 else e3`

`fst` `snd` `e=[]` `List.hd` `List.tl` `Some(e)` `None`

- Já sabemos como usar `let` locais

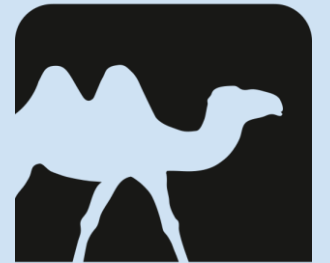
`let x = e in`

`let [rec] f ((x1:t1), ... (xN:tN)) = e in`

Até agora vimos

- Já sabemos como construir vários tipos

Vamos aprender a receita para
definirmos os nossos próprios tipos de dados em OCaml !



Enumerações e Árvores

Dica: os tipos são mais importantes para o desenho do programa do que para o algoritmo ...

`f(e1,e2,...,en) e1+e2 if e1 then e2 else e3`

`fst snd e=[] List.hd List.tl Some(e) None`

- Já sabemos como usar `let` locais

`let x = e in`

`let [rec] f ((x1:t1), ... (xN:tN)) = e in`

Anatomia dos tipos em OCaml

As linguagens geralmente fornecem três formas para definir tipos

- Duas formas de “combinar” tipos existentes t_1, t_2, \dots, t_N num novo tipo t
 - “Cada um dos” : um valor do tipo t contem valores de ***cada um dos*** tipos t_1, \dots, t_N
 - “De um dos” : um valor do tipo t contem valores ***de um dos*** tipos t_1, \dots, t_N
- Uma forma de os valores do tipo t conterem outros valores (mais pequenos) do tipo t
 - “Autoreferência” : um valor do tipo t pode conter outros valores do tipo t

Anatomia dos tipos em OCaml

As linguagens geralmente fornecem três formas para definir tipos

- Duas formas de “combinar” tipos existentes t_1, t_2, \dots, t_N num novo tipo t
 - “Cada um dos” : um valor do tipo t contem valores de ***cada um dos*** tipos t_1, \dots, t_N
 - “De um dos” : um valor do tipo t contem valores ***de um dos*** tipos t_1, \dots, t_N

A capacidade de aninhar e “misturar e combinar” torna estas formas de construção ainda mais poderosos!

Composicionais

Exemplos de tipos em OCaml

- Tuplos são *cada um dos* : `um int * bool` contem um `int` e um `bool`
- Opções são *de um dos* : `um int option` contem um `int` ou nenhum dado
- Listas são *autoreferências* para usar todas as formas de definir tipos:
 - Uma lista `int list` contem (um `int` e uma outra `int list`) ou nenhum dado
- O aninhamento permite-nos construir todo o tipo de tipos interessantes
 - `((int * bool) option * (string list list)) option list`

Exemplos de tipos em OCaml

- Tuplos são *cada um dos* : `um int * bool` contem um `int` e um `bool`
- Opções são *de um dos* : `um int option`
- Listas são *autoreferências* para usar todas
 - Uma lista `int list` contem (um `int` e um
- O aninhamento permite-nos construir todo o tipo de tipos interessantes
 - `((int * bool) option * (string list list)) option list`

Os tipos fornecem uma linguagem compacta para exprimir a “forma” dos dados!

Registos e Variantes



- **Registos**: uma outra forma de contruir tipo de cada-um-dos, um pouco diferente dos tuplos
 - Tipos com nome
 - Campos com nome
- **Variantes**: uma nova forma de construir os nossos tipos de-um-dos
 - Tipos com nome
 - Podemos definir um tipo cujo valor pode ser um `int` ou uma `string`
 - Para **construirmos** variantes, as definições do tipo incluem **construtores** como `Some`, `None`, `::`, `[]`
 - Para **usar** variantes, utilizamos a **expressão** `match` (**correspondência de padrões**) para testar e aceder a cada elemento

Tuplos Vs. Registos

- Semanticamente não são muito diferentes. Comparem:
 - `(1, true, "três")` vs. `{a=1; b=true; c="três"}`
 - Tuplos são um pouco mais curtos, enquanto os registos são mais fáceis de lembrar (documentação)
 - Evitem tuplos grandes e considerem registos quando vários campos têm o mesmo tipo
- Questão de conceção da linguagem: *Por posição* (**tuplos**); *por nome* (**registos**)
 - As funções fazem um pouco dos dois: *posição* para quem chama, mas *nome* para quem utiliza a função

Registos e Variantes



- **Registos**: uma outra forma de contruir tipos diferente dos tuplos
 - Tipos com nome
 - Campos com nome

Os variantes não são como os registos!

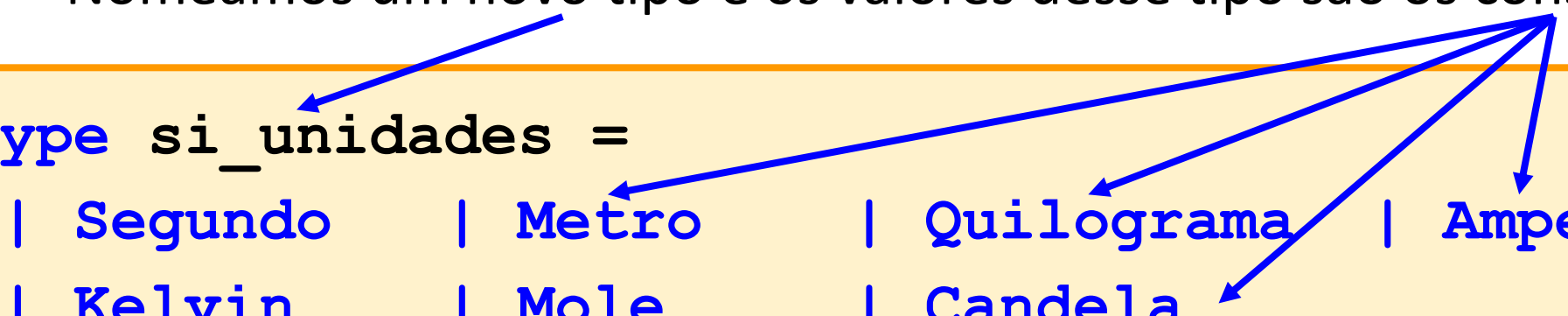
**O tipo de-um-dos não é o mesmo que
cada-um-dos!**



- **Variantes**: uma nova forma de construir os nossos tipos de-um-dos
 - Tipos com nome
 - Podemos definir um tipo cujo valor pode ser um `int` ou uma `string`
 - Para **construirmos** variantes, as definições do tipo incluem **construtores** como `Some`, `None`, `::`, `[]`
 - Para **usar** variantes, utilizamos a **expressão** `match` (**correspondência de padrões**) para testar e aceder a cada elemento

Enumerações

- Pode utilizar tipos variantes para *enumerar* um conjunto fixo de valores para um novo tipo
 - A utilização mais simples dos tipos de variantes, que são muito mais poderosos
 - Começamos com este caso de uso simples e aumentaremos rapidamente
- Nomeamos um novo tipo e os valores desse tipo são os construtores



```
type si_unidades =  
  | Segundo      | Metro      | Quilograma  | Ampere  
  | Kelvin       | Mole       | Candela     |
```

Definição do Variante

```
type si_unidades =  
  | Segundo      | Metro        | Quilograma   | Ampere  
  | Kelvin       | Mole         | Candela
```

- Sintaxe:
 - Primeiro | opcional
 - O construtor deve iniciar com letra maiúscula
- Semântica: Adiciona no ambiente o nome do tipo e o(s) construtor(s)

Construção

```
type si_unidades =  
  | Segundo      | Metro        | Quilograma   | Ampere  
  | Kelvin       | Mole         | Candela
```

- Estes 7 construtores são agora valores (e, portanto, expressões)
 - 7 *diferentes* formas de construir um valor de um novo tipo **si_unidades**
 - Não existem outros valores do tipo **si_unidades**
 - Por exemplo: [**Segundo**; **Metro**; **Segundo**] tem o tipo **si_unidades list**

Booleanos (novamente)

- Uma das enumerações mais simples é o nosso velho amigo booleano
- Poderia ser quase definido por `type bool = true | false`
- O problema é que quebra a regra da letra maiúscula (a linguagem quebra a sua própria regra)
 - Mas podemos escrever

`type booleano = Verdadeiro | Falso`

De qualquer modo, podemos agora definir enumerações e construir valores para elas, mas ainda precisamos de uma forma de utilizar os seus valores.

Alguma ideia ?

Usar Tipos Variante com Correspondência de Padrões (pattern-matching)

```
let string_of_si_unidades u =  
  match u with  
  | Segundo      -> "segundo"  
  | Metro        -> "metro"  
  | Quilograma   -> "quilograma"  
  | Ampere       -> "ampere"  
  | Kelvin       -> "kelvin"  
  | Mole         -> "mole"  
  | Candela      -> "candela"  
  
let s =  
  string_of_si_unidades Ampere
```

- Expressões match com um ramo para cada construtor
- Como se fosse um if encadeado, mas com um ramo para cada construtor
- Guia a verificação do tipo: todos os ramos devem ter o mesmo tipo e *cada construtor* deve ter *um ramo*
- Guia as regras de avaliação: avalia exatamente um ramo

Booleanos (novamente)

```
if e1 then e2 else e3
```

```
match e1 with  
| true  -> e2  
| false -> e3
```

- De facto, podemos usar expressões com `match` em vez de expressões condicionais
 - Sim, funciona em OCaml
- Em boa verdade, *não é um bom estilo*
 - Demonstra que uma funcionalidade (expressões condicionais) podem ser explicadas inteiramente em termos de uma outra funcionalidade mais poderosa (expressões `match`)
 - Outro exemplo de “**açúcar sintático**”

Para lá das enumerações

Os tipos variante em OCaml são muito poderosos, como vamos ver.

- Cada construtor pode transportar dados [ou não]
 - A definição de tipo indica o tipo de dados que cada construtor transporta
- A autoreferência é permitida na definição de tipo
 - Pode transportar “uma outra coisa igual”
- Vamos ver alguns exemplos
 - Um exemplo sem muito sentido para aprender a sintaxe e a semântica
 - Um exemplo para diferentes formas
 - Um exemplo que usa autoreferência para definir expressões aritméticas (em forma de árvores)

Definição de Variantes

```
type sem_muito_sentido =  
  | A of int * bool * string list  
  | F of string  
  | Pizza
```

Sintaxe: `type tname = | C1 [of t1] | ... | CN [of t1]`

- `t1`, ..., `tN` são tipos (e podem incluir o próprio `t`)
- `C1`, ..., `CN` são *construtores*
- Lembrem-se [...] é uma *meta sintaxe* para opcional

Adiciona um novo tipo (variante) `tname` e os construtores `C1`, ..., `CN`

- Os tipos estão no seu próprio namespace
- Os construtores estão no seu próprio namespace
- Contudo, pode *sobrepôr-se* aos tipos e construtores definidos anteriormente ⚠

Devem definir um tipo variante antes de o poderem *construir* ou *usar*

Construtores

```
type sem_muito_sentido =  
  | A of int * bool * string list  
  | F of string  
  | Pizza
```

Sintaxe: **C** e

Verificação de Tipo:

- Se **C** está no ambiente devido ao tipo **t = ... | C of tC | ...**
- E e tem tipo **tC** (deve providencia argumentos correto para **C**)
- Então **C** e tem tipo **t**
- Senão, não tipa

Avaliação: Avaliar e num valor **v** e **C v** é um valor

- “Marca (tag) os dados **v** com um **C**”, produzindo um valor do tipo **t**

Expressões match

Uma forma de expressão poderosa e concisa que combina três coisas:

1. Um condicional múltiplo que se ramifica com base na variante de um tipo de-um-dos que tem
2. Extrai os dados subjacentes que foram “marcados” com um construtor
3. Liga os dados extraídos a variáveis locais que podem depois ser utilizadas no ramo condicional escolhido

Isto é elegante (depois de nos habituarmos) e evita erros como o esquecimento de casos ou quando tentamos extrair os dados “errados” para o marcador (tag) que temos

Usar Variantes

[generalização em breve!]

Sintaxe:

```
match e with P1 -> e1 | ... | PN -> eN
```

- Onde `e`, `e1`, ..., `eN` são expressões
- Onde `P1`, ..., `PN` são *padrões*

Por agora, cada padrão deverá ser um dos seguintes:

- `C` se `C` é um construtor que não transporta dados
- `C (x1,x2,...,xn)` se `C` é um construtor que transporta um tuplo
- `C x` caso contrário

onde `x1`, ..., `xn` e `x` são *variáveis*

```
type sem_muito_sentido =  
  | A of int * bool * string list  
  | F of string  
  | Pizza
```

Padrões

- Os padrões não são expressões, apesar de se parecerem um pouco com elas
- São utilizadas para estabelecer *correspondências* e para *ligar variáveis locais quando estas correspondem*

Tipos Variante Recursivos

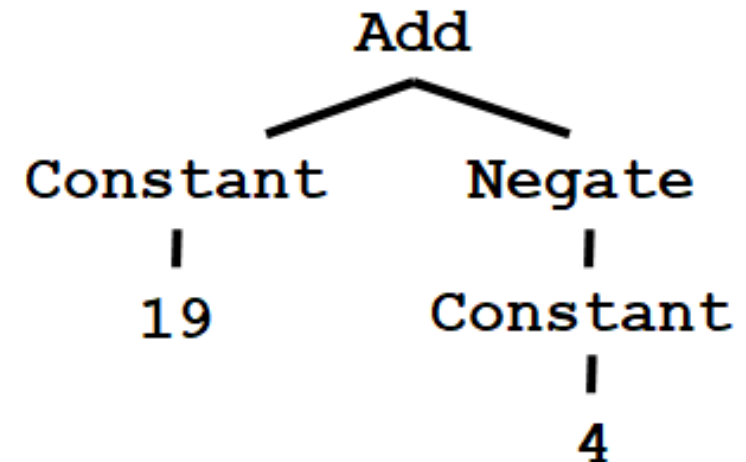
Forma concisa e elegante de definir diferentes tipos de árvores

- Os nós são marcados com um construtor com filhos para os dados transportados

```
type expr =  
  | Constant of int  
  | Negate of expr  
  | Add of expr * expr  
  | Mul of expr * expr
```

As funções sobre o tipo `expr` tipicamente usam recursividade sobre a estrutura em árvore

```
Add(Constant 19,  
      Negate(Constant 4))
```



Definição de opções

Utilizando o que já sabemos:

```
type int_option = NoInt | OneInt of int
```

Utilizando a nova funcionalidade para definir o nosso próprio *construtor de tipos*

```
type 'a myoption = MyNone | MySome of 'a
```

Exatamente como as opções “built-in” são definidas (apenas na “standard library”):

```
type 'a option = None | Some of 'a
```


Então **usem** a Correspondência de Padrões

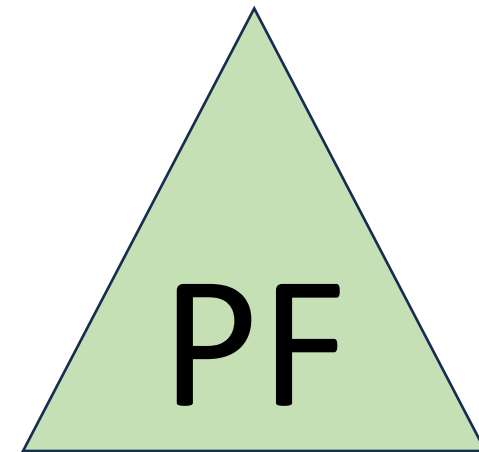
- None e Some são (apenas) construtores para 'a option
- *Construir* com construtores e *usar* com expressões match
 - Não usem Option.get e Option.is_some daqui para a frente (⚠)
 - Tirem partido das vantagens dos tipos de variantes e da correspondência de padrões



**Correspondência
de Padrões**



Recursividade



Imutabilidade

Como as linguagens definem os tipos variantes

- O OCaml incorporou uma poderosa correspondência de padrões na linguagem e utilizou-a como a primitiva fundamental para utilizar tipos de-um-dos
- Outras linguagens fizeram outras escolhas.
 - Por exemplo, o OCaml poderia ter optado por:
 - Simplesmente manter as definições de tipos de variantes como vimos
 - Ter introduzido *funções* como `isEmpty`, `hd` e `tl` na linguagem
 - Muitas outras opções (as linguagens são feitas de escolhas, uma vez que não podemos ter “tudo” numa só)



Correspondência de Padrões

O OCaml também tem correspondência de padrões para cada-um-dos tipos

- O padrão (x, y, z) corresponde a triplos $(v1, v2, v3)$
 - De forma similar para qualquer tuplo, de qualquer tamanho
 - Padrões semelhantes para registos $\{f1=x1; \dots; fn=xn\}$

Útil para *padrões aninhados* (veremos mais à frente)

Mas também é útil para fazer com que as expressões let *sejam mais poderosas*

O que não devem fazer

- As expressões de correspondência de um ramo fazem sentido semântico mas não em estilo

```
let soma_triplo tr =  
  match tr with  
  | (x,y,z) -> x+y+z
```

Nova funcionalidade

A sintaxe da expressão `let` é de facto `let p = e1 in e2`

- Onde `p` é um padrão

(Reparem que `let p = e1` (Top-level) também é válido)

É ótimo para extrair vários dados de uma só vez

- Não precisamos mais de `fst`, `snd`, `fst3`, `snd3`, `thd3`, ...

```
let soma_triplo tr =  
  let (x,y,z) = tr in  
  x+y+z
```

Mais uma nova funcionalidade

Os argumentos de funções também podem utilizar padrões

- Semântica da chamada da função: correspondência de padrões no argumento extrai dados para o corpo da função

```
let soma_triplo tr =  
  let (x,y,z) = tr in  
  x+y+z
```

```
let soma_triplo (x,y,z) =  
  x+y+z
```

```
let dez = soma_triplo (5,2,3)
```

As funções em OCaml

Uma função que recebe um triplo e soma os seus componentes inteiros:

```
let soma_triplo (x,y,z) =  
    x+y+z
```

Uma função que recebe três inteiros e efetua a sua soma:

```
let soma_triplo (x,y,z) =  
    x+y+z
```

Conseguem ver a diferença ? (Eu também não.) 😊

As funções em OCaml

Recebem exatamente um argumento

Podemos *simular* múltiplos argumentos com tuplos e padrões de tuplos

- Elegante, composicional

Por vezes, permite conveniência

- Devolver um tuplo de uma função e passá-la como vários argumentos para uma outra

Veremos mais à frente que existe uma forma diferente e mais comum de simular múltiplos argumentos que *também* tem suporte sintático incorporado na linguagem

Padrões Aninhados

Padrões Aninhados

- Podemos aninhar padrões dentro de outros padrões
 - Tal como podemos aninhar expressões em profundidade tanto quanto quisermos
 - Em qualquer lugar onde uma variável possa aparecer nos nossos padrões atuais, podemos colocar um padrão
- Assim, o significado completo de “pattern-matching” é comparar um padrão com um valor para a “mesma forma” e associar variáveis às “partes corretas”
 - Definição recursiva mais precisa após mais alguns exemplos

Exemplos

Vejam os exemplos de código para vários exemplos de utilização de padrões aninhados para exprimir algoritmos de forma concisa e elegante

Alguns usos diferentes:

- Correspondência de várias coisas com a mesma forma ao mesmo tempo
- Entrar em vários níveis da estrutura de dados de uma só vez
- Devem utilizar “wildcards” _ quando não precisarem dos dados
- Criem expressões de correspondência que se assemelhem a tabelas

Expressões `match` aninhadas?

- Por vezes, uma expressão de correspondência dentro de uma expressão de correspondência é uma oportunidade perdida para a correspondência de padrões aninhados

```
match xs with  
| [] -> 0  
| x::xs' -> match xs' with ...
```

- Outras vezes é necessário porque é preciso calcular primeiro com os dados extraídos através da correspondência externa

```
match xs with  
| [] -> 0  
| x::xs' -> match f xs' with ...
```

Expressões match gerais

Sintaxe:

match e with $P1 \rightarrow e1 \mid \dots \mid PN \rightarrow eN$

let $P1 = e$ in $e1$

let [rec] $f \ P1 = e1$

Onde:

- $e, e1, \dots, eN$ são expressões
- $P1, \dots, PN$ são padrões

Expressões match gerais

Sintaxe:

match e with $P1 \rightarrow e1 \mid \dots \mid Pn \rightarrow en$

let $P1 = e$ in $e1$

let [rec] $f \ P1 = e1$

Onde:

- $e, e1, \dots, en$ são expressões
- $P1, \dots, Pn$ são padrões

Padrões **NÃO** são expressões!
(embora pareçam expressões)

Sintaxe: Um padrão pode ser

- x : uma variável
- $_$: um “wildcard”
- $(p1, \dots, pn)$: um tuplo de padrões pi
- $C \ p$: um construtor C aplicado ao padrão p

Exemplos

- O padrão $\mathbf{a :: b :: c :: d}$ faz correspondência a todas as listas com mais de (\geq) 3 elementos
- O padrão $\mathbf{a :: b :: c :: []}$ faz correspondência a todas as listas com exatamente 3 elementos
- O padrão $\mathbf{((a, b) , (c, d)) :: e}$ faz correspondência a todas as listas não vazias de pares de pares

Semântica das expressões `match`

```
match e with P1 -> e1 | ... | PN -> eN
```

1. Avaliar a expressão `e` em `v`
2. Verificar `P1, P2, ..., PN` por ordem; procurar o primeiro `Pi` que faz correspondência
3. Avaliar `ei` no ambiente estendido com os “bindings” do match
4. O resultado de (3) é o resultado global

Nota: Isto é a *semântica*, mas a implementação pode *otimiza-la com conceitos de procura binária*

Verificação de Tipo das expressões `match`

```
match e with P1 -> e1 | ... | PN -> eN
```

- Verificar se a expressão `e` tem algum tipo `t`
- Cada `Pi` deve corresponder a algum valor do tipo `t`
- Padrão similar na definição de correspondência de tipos para obter os tipos de ligações
 - Verificação do tipo de `ei` no ambiente estático que contêm os “bindings”
- Todas as expressões `e1,...,en` devem ter o mesmo tipo `t2`, que é do tipo geral

Ainda temos mais algumas funcionalidades interessantes:

- Um erro se um `Pi` nunca tiver uma correspondência devido aos padrões anteriores (código morto)
- Um aviso caso um valor do tipo `t` não possa corresponder a `Pi` (dizemos que é uma correspondência incompleta)

Até agora

- Temos praticamente “tudo” o que precisamos
 - Funções de ordem superior
 - Registos e Tuplos
 - Tipos de dados recursivos (Listas e Árvores)
- Alguns pormenores importantes para a próxima aula
 - Recursividade Terminal
 - Exceções
- Depois, módulos (simples) ...

Créditos para Dan Grossman.