Linguagens e Ambientes de Programação (2018/2019)

Teórica 04 (18/mar/2019)

Tipos soma (uniões). Os tipos somas árvore binária e árvore n-ária.
Padrões. Padrões disjuntos. Emparelhamento de padrões.

Tipos soma (uniões)

Muitas linguagens de programação incluem uma construção específica para a definição de tipos cujos valores podem assumir **diferentes variantes**, disjuntas entre si. Essa construção designa-se genericamente por **tipo soma**.

Por exemplo, numa linguagem com suporte para tipos soma é possível definir um tipo de dados *cshape*, para representar formas geométricas coloridas cujos valores podem assumir as três seguintes variantes: *Line*, *Circle* e *Rect*.

Um tipo soma permite conciliar o **geral** com o **particular**. Por um lado os elementos das variantes *Line*, *Circle* e *Rect* são formas geométricas coloridas com algumas propriedades comuns: no mínimo todas têm um cor! Por outro lado são variantes, cada uma delas com dados específicos associados: um círculo tem um centro e um raio, uma linha tem dois pontos extremos, etc.

Os tipos soma do Pascal chamam-se *registos com variante*.
Os tipos soma do C são as *uniões*.
Os tipos soma do Java, Smalltalk e C++ são as *classes abstratas* (imagine uma classe abstrata *cshape* com três subclasses concretas *Line*, *Circle* e *Rect*). [Em rigor, as classes abstratas são um bocadinho diferentes dos tipos soma: os tipos soma são entidades fechadas enquanto que as classes abstratas são extensíveis.]

Como é em OCaml?

Tipos soma em OCaml

Para exemplificar, o tipo soma `cshape` atrás referido pode ser definido em OCaml da seguinte forma, usando os tipos auxiliares color e point:

```
type color = int ;;
type point = float*float ;;

type cshape = Line of color*point*point
            | Circle of color*point*float
            | Rect of color*point*point ;;
```

Repare que o nome dos tipos definidos pelo utilizador começa sempre por letra minúscula e o nome de cada variante começa sempre por letra maiúscula. Chamam-se **etiquetas**, aos nomes das variantes.

Continuando a usar o mesmo exemplo, vejamos agora quais são os mecanismos essenciais para escrever e manipular valores de tipos soma.

Literais: Eis dois literais de tipo cshape. Repare como o nome de cada variante é usado para marcar os respetivos literais.

```
let a = Line (34658, (2.5, 7.8), (-24.005, 1000.0001)) ;;
let b = Circle (11111, (-24.005, 1000.0003), 3.1233333) ;;
```

Construção: Por exemplo, a seguinte função cria círculos centrados no ponto zero:

```
let zeroCircle c r = Circle (c, (0.0, 0.0), r) ;;
```

Processamento: Eis uma função que calcula a área duma forma colorida. Os elementos de qualquer tipo soma são processados usando emparelhamento de padrões.

```
let area cs =
    match cs with
        Line   (_, _, _) -> 0.0
      | Circle (_, _, r) -> 3.14159 *. r *. r
      | Rect   (_, (tx,ty), (bx,by)) -> abs_float ((bx -. tx) *. (by -. ty))
;;
```

Eis uma função que determina o raio duma forma. Só está definida para círculos.

```
let radius (Circle (_, _, r)) = r ;;
```

Alguns tipos soma predefinidos em OCaml

1. O tipo **'a list** é um tipo soma. Internamente a sua definição assemelha-se a:

   ```
   type 'a list = Nil | Node of 'a * 'a list ;;
   ```

2. O tipo **bool** também é um tipo soma, internamente é definido da seguinte forma:

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

   O OCaml abre uma exceção neste caso, e permite que o nome das variantes comece por letra minúscula.

3. O tipo **'a option** é um tipo soma que permite representar o conceito de **ausência de valor**, em situações que tal possa ser útil. Internamente, é definido assim:

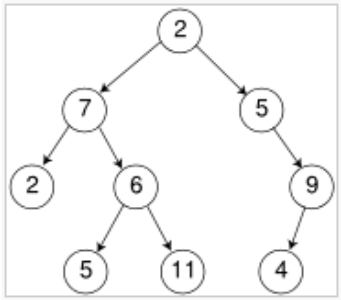
   ```
   type 'a option = None | Some of 'a ;;
   ```

Os três papéis das etiquetas dum tipo soma

As etiquetas dum tipo soma tem três papéis:

- Na definição do tipo, Identificam os vários ramos.
- Denotam os *construtores* que o sistema cria automaticamente para o tipo em causa. Um *construtor* é uma função especial que gera valores dum tipo soma. Quando escrevemos Circle(111, (0.0, 0.0), 12.4) estamos a chamar um construtor das nossas formas.
- São elementos constituintes de novos padrões que a linguagem passa a reconhecer automaticamente.

Árvores binárias



Em programação, as árvores binárias permitem exprimir informação hierarquizada e permitem organizar dados por forma a aumentar a velocidade de acesso a eles.

O tipo soma **árvore binária** não está predefinido em OCaml, mas é fácil de definir usando um tipo soma:

```
type 'a tree = Nil | Node of 'a * 'a tree * 'a tree ;;
```

Literais: Eis uma constante do tipo `int tree`:

```
let t = Node(1, Node(2,Nil,Nil), Node(3,Nil,Nil)) ;;
```

Construção: Por exemplo, a seguinte função cria folhas da árvore:

```
let makeLeaf v = Node(v, Nil, Nil) ;;
```

Processamento: Eis uma função que determina o número total de nós duma árvore binária:

```
let rec size t =
    match t with
        Nil -> 0
```

```
        | Node(x,l,r) ->
              1 + size l + size r
;;
```

Eis um exemplo de avaliação duma expressão envolvendo uma chamada da função "size":

```
  size (Node(1, Node(2,Nil,Nil), Node(3,Nil,Nil)))
= 1 + size (Node(2,Nil,Nil)) + size (Node(3,Nil,Nil))
= 1 + (1 + size Nil + size Nil) + (1 + size Nil + size Nil)
= 1 + (1 + 0 + 0) + (1 + 0 + 0)
= 1 + 1 + 1
= 3
```

Vocabulário relativo a árvores

- Raiz - nó sem ascendentes
- Nó interno - nó diferente da raiz com pelo menos um filho
- Folha - nó sem filhos

Método indutivo aplicado a árvores binárias

No caso das árvores binárias, o método indutivo consiste na seguinte técnica:

- Para programar o caso geral G duma função recursiva, assume-se como PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR que os resultados L, R de aplicar a própria função a argumentos *mais simples do que os iniciais* (por exemplo às duas subárvores) já se encontram disponíveis. O problema que é então preciso resolver é o seguinte: COMO É QUE A PARTIR DE L E R SE CHEGA A G?

Exemplo de utilização do método indutivo

Problema: Programar uma função "size" que determine o número total de nós duma árvore binária:

Resolução: A função "size" aplica-se a uma árvore, a qual será, naturalmente, processada usando emparelhamento de padrões. Aplicando o método indutivo ao caso geral "G=Node(x,l,r)", vamos começar por assumir que já temos o problema resolvido para os casos "L=size l" e "R=size r". Obtemos assim o seguinte PONTO DE PARTIDA PARA COMEÇAR A RACIOCINAR:

```
 let rec size t =
    match t with
         Nil ->  ...
       | Node(x,l,r) ->
            ... size l ... size r ...
;;
```

O problema que temos para resolver é o seguinte: *Como é que se obtém o valor de "G=size (Node(x,l,r))" a partir dos valores "L=size l" e "R=size r"?* Mas este problema é simples. De facto, basta adicionar uma unidade a L+R para se obter G!

Preenchendo o que falta no esquema anterior, surge a solução final:

```
let rec size t =
    match t with
        Nil -> 0
      | Node(x,l,r) ->
            1 + (size l) + (size r)
;;
```

Mais duas funções sobre árvores binárias

Altura duma árvore:

```
(* height: 'a tree -> int *)

let rec height t =
    match t with
       Nil -> 0
```

```
      | Node(x,l,r) ->
            1 + max (height l) (height r)
;;
```

Árvore ao espelho:

```
(* mirror: 'a tree -> 'a tree *)

let rec mirror t =
    match t with
      Nil -> Nil
    | Node (x,l,r) ->
          Node (x,mirror r,mirror l) (* o r e l trocam de posição *)
;;
```

Árvores n-árias

Numa árvore n-ária, cada nó pode ter um número qualquer (ilimitado) de filhos.

O tipo soma **árvore n-ária** pode definir-se assim em OCaml:

```
type 'a ntree = NNil | NNode of 'a * 'a ntree list ;;
```

Repare que enquanto numa folha duma árvore binária se usam dois Nil para indicar que não há filhos, numa folha duma árvore n-ária usa-se uma lista vazia para indicar a mesma coisa.

Literais: Eis uma constante de tipo "int ntree":

```
let nt = NNode(1, [NNode(2,[]); NNode(3,[]); NNode(4,[])]) ;;
```

Construção: Por exemplo, a seguinte função cria folhas:

```
let makeLeaf v = NNode(v, []) ;;
```

Processamento: A função `size` determina o número de nós duma árvore n-ária. Note que se usa uma função auxiliar que processa uma lista de árvores.

```
(* size: 'a ntree -> int *)

let rec size t =
    match t with
        NNil -> 0
      | NNode(x,cs) -> 1 + lsize cs
and lsize tl =
    match tl with
        [] -> 0
      | t::ts -> size t + lsize ts
;;
```

Esquema geral da utilização do método indutivo no tratamento de árvores n-árias:

```
let rec f t =
    match t with
        NNil -> ...
      | NNode(x,cs) -> ... lf cs ...
and lf tl =
    match tl with
        [] -> ...
      | t::ts -> ... f t ... lf ts ...
;;
```

Árvore ao espelho:

```
(* mirror: 'a ntree -> 'a ntree *)

let rec mirror t =
    match t with
        NNil -> NNil
```

```
      | NNode(x,cs) -> NNode(x,lmirror cs)
and lmirror tl =
    match tl with
        [] -> []
      | t::ts -> lmirror ts @ [mirror t]
;;
```

Invariante: A representação das árvores n-árias é ambígua pois a mesma árvore pode ser apresentada de várias formas. Isto acontece porque ocorrências de `NNil` numa lista de filhos não acrescentam nada de útil. Por exemplo, as seguintes árvores n-árias são equivalentes e representam sempre a mesma folha:

```
NNode(5, [])
NNode(5, [NNil])
NNode(5, [NNil; NNil])
NNode(5, [NNil; NNil; NNil])
```

Para evitar esta ambiguidade, vamos introduzir o seguinte invariante:

- O literal `NNil` representa a árvore vazia isolada e proibimos que `NNil` ocorra como parte de outra árvore.

Para ajudar a cumprir o invariante, introduzimos a função auxiliar `ncons`. A função acrescenta uma árvore n-ária à cabeça duma lista de árvores n-árias apenas se a árvore não for vazia:

```
(* ncons : 'a ntree -> 'a ntree list -> 'a ntree list *)

let ncons t l =
    if t = NNil then l  (* ignora NNil *)
    else t::l           (* acrescenta *)
;;
```

Um exemplo que usa `ncons`: Eliminar duma árvore `t`, todas as subárvores com um dado valor `v` na raiz:

```
(* delete: 'a -> 'a ntree -> 'a ntree *)

let rec delete v t =
    match t with
        NNil -> NNil
      | NNode(x,cs) -> if x = v then NNil
                       else NNode(x,ldelete v cs)
and ldelete v tl =
    match tl with
        [] -> []
      | t::ts -> ncons (delete v t) (ldelete v ts)
;;
```

Padrões

Um **padrão** é uma expressão especial que representa um **conjunto de valores**. A sintaxe dum padrão fornece geralmente uma boa intuição sobre a estrutura dos valores em causa.

A utilização de padrões torna as funções mais fáceis de escrever e de entender. Os padrões são, portanto, bons amigos do/a programador/a. As linguagens funcionais antigas (e.g. Lisp) não usavam padrões, mas as linguagens funcionais modernas (e.g. OCaml, Haskell) não os dispensam.

Exemplos de padrões:

```
Padrão          Conjunto de valores representados
~~~~~~          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[]              lista vazia
[x]             listas com um elemento
[x;y]           listas com dois elementos
x::xs           listas não vazias
x::y::xs        listas com pelo menos dois elementos
5::xs           listas cujo primeiro elemento é 5
x               todos os valores (padrão universal)
_               padrão universal anónimo
(x,y)           todos os pares ordenados
(0,y)           todos os pares ordenados cuja 1ª componente é 0
```

```
8               inteiro 8
(x,y)::xs       lista não vazia de pares ordenados
'a'..'z'        letras de 'a' a 'z'
```

Numa expressão **match** podem ocorrer padrões em número variável. Durante a avaliação, esses padrões são explorados sequencialmente, de cima para baixo.

Por exemplo, o que faz a seguinte função?

```
let rec count5 l =
    match l with
        [] -> 0
      | 5::xs -> 1 + count5 xs
      | _::xs -> count5 xs
;;
```

E o que faz esta outra função?

```
let rec count5x l =
    match l with
        [] -> 0
      | _::xs -> count5x xs
      | 5::xs -> 1 + count5x xs
;;
```

Onde podem ocorrer os padrões?

Na sintaxe do OCaml, está previsto que os padrões ocorrem nos seguintes contextos:

- Atrás das setas das expressões **match**:

  ```
  let rec len l =
      match l with
          [] -> 0
        | _::xs -> 1 + len xs
  ;;
  ```

- No sítio dos argumentos das funções:

  ```
  fun (x,y) -> (y,x)

  let f (x,y) =
      (y,x)
  ;;
  ```

- Atrás do sinal de = das expressões **let-in**:

  ```
  let (x,y) = g 0 in
      x + y
  ;;
  ```

Atenção ao contexto!

- Nos contextos atrás indicados, as expressões são interpretadas como padrões. Por exemplo, num desses contextos, **x::xs** representa o conjunto de todas as listas não vazias.

- Fora dos contextos anteriores, as expressões são interpretadas da forma habitual. Por exemplo, fora desses contextos, **x::xs** representa a aplicação do construtor "cons" aos nomes x e xs.

Padrões disjuntos

Dois padrões dizem-se **disjuntos** se os conjuntos que eles representam são disjuntos. Numa expressão **match**, a ordem dos casos só é irrelevante se os padrões forem disjuntos entre si.

Exemplos:

- Os dois padrões que ocorrem na função `len` **são disjuntos**:

```
let rec len l =
     match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

- Os dois padrões que ocorrem na função `fact` **não são disjuntos**:

```
let rec fact n =
     match n with
       0 -> 1
     | n -> n * fact (n-1)
;;
```

Emparelhamento

Como resultado do **emparelhamento dum valor com um padrão**, há duas consequências possíveis:

1. Falhanço;
2. Sucesso, e os nomes que ocorrem no padrão ficam ligados a valores.

Restrições

- Num padrão, não se permite a repetição de nomes. Por exemplo o padrão `(x,x)::xs` é inválido.
- Todos os nomes que ocorrem num padrão são considerados nomes novos, mesmo que esses nomes ocorram no ambiente envolvente.

A seguinte função aplica-se a uma lista de pares ordenados, e conta o número de pares com as duas componentes iguais:

```
let rec eqPairs l =
    match l with
      [] -> 0
    | (x,y)::xs ->    (* o padrão (x,x)::xs parece melhor, mas seria inválido *)
        if x=y then 1 + eqPairs xs
        else eqPairs xs
;;
```

A seguinte função aplica-se a uma lista de pares ordenados, e conta o número de pares em que a segunda componente é igual a um valor dado:

```
let rec countPairs v l =
    match l with
      [] -> 0
    | (x,y)::xs ->    (* o padrão (x,v)::xs parece melhor, mas não interessa pois contém um v "novo" *)
        if y=v then 1 + countPairs v xs
        else countPairs v xs
;;
```

#120