Estruturas de dados

Pedro Vasconcelos

14 de Abril de 2020

Estruturas de dados

Nesta aula vamos extender a linguagem FUN com estruturas de dados:

- pares e tuplos;
- variantes;
- *records*.

Bibliografia: Capítulo 3 de *Programming languages*, Mike Grant e Scott Smith.
`http://www.cs.jhu.edu/~scott/pl/book`

Pares

- Combinação de dois valores
- Noção matemática de producto cartesiano

$$a \in A,\ b \in B \implies (a, b) \in A \times B$$

- Duas projecções para extrair as componentes

$$\text{fst} : A \times B \to A \qquad \text{snd} : A \times B \to B$$
$$\text{fst}\ (x, y) = x \qquad \text{snd}\ (x, y) = y$$

Pares: extensões à linguagem

Acrescentamos um construtor e duas funções de projeção:

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \textbf{pair}\ e_1\ e_2 & \text{construtor} \\
 & | & \textbf{fst}\ e & \text{projeção 1} \\
 & | & \textbf{snd}\ e & \text{projeção 2}
\end{array}
$$

Pares: extensões à semântica operacional

Acrescentamos pares ao conjunto de valores:

$$v \ ::= \ \cdots \ \mid \ \textbf{pair}\ v_1\ v_2$$

Três novas regras:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\textbf{pair}\ e_1\ e_2 \Downarrow \textbf{pair}\ v_1\ v_2}$$

$$\frac{e \Downarrow \textbf{pair}\ v_1\ v_2}{\textbf{fst}\ e \Downarrow v_1} \qquad \frac{e \Downarrow \textbf{pair}\ v_1\ v_2}{\textbf{snd}\ e \Downarrow v_2}$$

Pares: extensões à SECD

- Acrescentamos pares à memória *Store*.
- Três novas instruções:

$$\langle v_2 : v_1 : s,\ e,\ \text{PAIR} : c,\ d,\ m\rangle \longrightarrow \langle a : s,\ e,\ c,\ d,\ m'\rangle$$
$$\text{onde} \quad a = \text{next } m$$
$$m' = m[a \mapsto (v_1, v_2)]$$

$$\langle a : s,\ e,\ \text{FST} : c,\ d,\ m\rangle \longrightarrow \langle v_1 : s,\ e,\ c,\ d,\ m\rangle$$
$$\text{se } m[a] = (v_1, v_2)$$

$$\langle a : s,\ e,\ \text{SND} : c,\ d,\ m\rangle \longrightarrow \langle v_2 : s,\ e,\ c,\ d,\ m\rangle$$
$$\text{se } m[a] = (v_1, v_2)$$

- Exercício: modificar o compilador e interpretador.

Tuplos

$$(e_1, e_2, \ldots, e_n)$$

- $n = 0$: tuplo vazio (*unit*)
- $n = 1$: N/A
- $n = 2$: pares
- $n > 2$: trios, quartetos...

Tuplos

- Podem ser codificados usando pares, e.g.:

$$(e_1, e_2, \ldots, e_n) \equiv (e_1, (e_2, \ldots (e_{n-1}, e_n) \ldots))$$

- Optimização: usar representações directas para tamanhos comuns, e.g. em Haskell 98:
  - construtores definidos pelo menos até $n = 15$;
  - acesso usando encaixe de padrões;
  - projeções pré-definidas apenas para pares.

Listas

Tal como no cálculo-λ, podemos codificar listas usando pares:

$$
\begin{aligned}
[\,] &\equiv \textbf{pair}\ 1\ 0 \\
(:) &\equiv \lambda x.\, \lambda y.\, \textbf{pair}\ 0\ (\textbf{pair}\ x\ y) \\
\textbf{null} &\equiv \lambda x.\, \textbf{fst}\ x \\
\textbf{head} &\equiv \lambda x.\, \textbf{fst}\ (\textbf{snd}\ x) \\
\textbf{tail} &\equiv \lambda x.\, \textbf{snd}\ (\textbf{snd}\ x)
\end{aligned}
$$

- $(1, \ldots)$ representa a lista vazia.
- $(0, (x, y))$ representa a lista $x : y$.
- Codificação só funciona numa linguagem não tipada.
- Alternativa: usar variantes.

Variantes

- União de alternativas etiquetadas por construtores.
- Em Haskell:

$$\text{data } T \;=\; \ell_1\ t_1 \;\mid\; \ell_2\ t_2 \;\mid\; \cdots \;\mid\; \ell_n\ t_n$$

$\ell_1, \ell_2, \ldots, \ell_n$ etiquetas

$T, t_1, t_2, \ldots, t_n$ tipos

- Seleção feita usando *encaixe de padrões*.

Variantes: extensões à linguagem

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & | & \textbf{cons}\ \ell(e) & \text{construtor} \\
 & | & \textbf{case}\ e_0\ \textbf{of} & \text{seleção} \\
 & & \quad \ell_1(x_1) \to e_1 & \\
 & & \mid \ell_2(x_2) \to e_2 & \\
 & & \quad \vdots & \\
 & & \mid \ell_n(x_n) \to e_n &
\end{array}
$$

- As alternativas não têm de ser exaustivas.
- Padrões simples: a ordem não é importante.

Enumerações

- Exemplo (em Haskell): dias da semana

  $$\text{data Weekday} = \text{Mon} \mid \text{Tue} \mid \text{Wed} \mid \text{Thu} \mid \text{Fri} \mid \text{Sat} \mid \text{Sun}$$

- Variante com uma etiqueta por cada dia

  $$\textbf{cons}\ \text{Mon}(0), \textbf{cons}\ \text{Tue}(0), \ldots, \textbf{cons}\ \text{Sun}(0)$$

- Os valores associados não são relevantes (e.g. zero)

Uniões

Opção entre duas alternativas (em Haskell):

$$\text{data Either } a\ b\ =\ \text{Left } a\ \mid\ \text{Right } b$$

$$\text{either}\ ::\ (a \to c) \to (b \to c) \to \text{Either } a\ b \to c$$
$$\text{either } f\ g\ (\text{Left } x) = f\ x$$
$$\text{either } f\ g\ (\text{Right } y) = g\ y$$

Na linguagem FUN:

$$\text{either} \equiv \lambda f.\, \lambda g.\, \lambda u.\, \textbf{case}\ u\ \textbf{of}\ \text{Left}(x) \to f\ x \mid \text{Right}(y) \to g\ y$$

Listas

Codificar listas usando variantes e pares:

$$
\begin{aligned}
[\,] &\equiv \textbf{cons}\ \mathsf{nil}(0) \\
(:) &\equiv \lambda x.\,\lambda y.\,\textbf{cons}\ \mathsf{list}(\textbf{pair}\ x\ y) \\
\textbf{null} &\equiv \lambda x.\,\textbf{case}\ x\ \textbf{of}\ \ \mathsf{nil}(\_) \to 1 \\
&\qquad\qquad\qquad\qquad\quad \mathsf{list}(y) \to 0 \\
\textbf{head} &\equiv \lambda x.\,\textbf{case}\ x\ \textbf{of}\ \mathsf{list}(y) \to \textbf{fst}\ y \\
\textbf{tail} &\equiv \lambda x.\,\textbf{case}\ x\ \textbf{of}\ \mathsf{list}(y) \to \textbf{snd}\ y
\end{aligned}
$$

Dois construtores: nil para lista vazia e list para lista não-vazia.

Variantes: extensões à semântica operacional

$$v \ ::= \ \cdots \ \mid \ \mathbf{cons}\ \ell(v)$$

$$\frac{e \Downarrow v}{\mathbf{cons}\ \ell(e) \Downarrow \mathbf{cons}\ \ell(v)}$$

$$\frac{e \Downarrow \mathbf{cons}\ \ell_j(v_j) \qquad e_j[v_j/x_j] \Downarrow v}{\mathbf{case}\ e\ \mathbf{of} \left\{ \begin{array}{l} \ell_1(x_1) \to e_1 \\ \vdots \\ \mid \ell_j(x_j) \to e_j \\ \vdots \\ \mid \ell_n(x_n) \to e_n \end{array} \right\} \Downarrow v}$$

Variantes: extensões à SECD

CONS ℓ constroi valor etiquetado ℓ (na memória *Store*).

MATCH $(\ell_1, c_n), \ldots, (\ell_n, c_n)$ generalização da instrução SEL:

1. inspeciona o valor no topo da pilha: (ℓ_j, v_j);
2. acrescenta v_j ao ambiente;
3. seleciona código c_j associado à etiqueta ℓ_j;
4. guarda a continuação na pilha dump

$$\langle v : s,\ e,\ \text{CONS}\ \ell : c,\ d,\ m\rangle \longrightarrow \langle a : s,\ e,\ c,\ d,\ m[a \mapsto (\ell, v)]\rangle$$
$$\text{onde}\ a = \text{next}\ m$$

$$\langle a : s,\ e,\ \text{MATCH}\ alts : c,\ d,\ m\rangle \longrightarrow \langle s,\ v_j : e,\ c_j,\ ([], [], c) : d,\ m\rangle$$
$$\text{onde}\ \ m[a] = (\ell_j,\ v_j)$$
$$(\ell_j, c_j) \in alts$$

Exercício: modificar o compilador e interpretador.

Projeções *vs.* expressões-*case*

Função recursiva para o comprimento de uma lista

Projeções

$$\textbf{fix}\ \lambda f.\,\lambda x.\,\textbf{if}\ (\textbf{null}\ x)\ \textbf{then}\ 0\ \textbf{else}\ 1 + f\ (\textbf{tail}\ x)$$

Expressão-case

$$\begin{aligned}\textbf{fix}\ \lambda f.\,\lambda x.\,\textbf{case}\ x\ \textbf{of}\quad & \mathsf{nil}(\_) \to 0 \\ \mid\ & \mathsf{list}\ (y) \to 1 + f(\textbf{snd}\ y)\end{aligned}$$

A expressão-*case* torna explícita a recorrência estrutural e evita a necessidade de construir um valor boleano.

Encaixe de padrões geral

- Múltiplas equações com padrões podem ser convertidas numa única definição usando expressões-*case*.
- Padrões embricados podem ser convertidos em múltiplas expressões-*case*.

Efficient Compilation of Pattern-Matching, P. Wadler. Capítulo 5 de *The Implementation of Functional Programming Languages*, S. L. Peyton Jones.

Exemplo

Último elemento duma lista:

```
last [x] = x
last (x:xs) = last xs
```

Transformação 1:

```
last xs = case xs of
            (x:[]) -> x
            (x:xs') -> last xs'
```

Transformação 2:

```
last xs = case xs of
            (x:xs') -> case xs' of
                         [] -> x
                         (x':xs'') -> last xs'
```

Exemplo

Último elemento duma lista:

```
last [x] = x
last (x:xs) = last xs
```

Transformação 1:

```
last xs = case xs of
            (x:[]) -> x
            (x:xs') -> last xs'
```

Transformação 2:

```
last xs = case xs of
            (x:xs') -> case xs' of
                         [] -> x
                         (x':xs'') -> last xs'
```

Exemplo

Último elemento duma lista:

```
last [x] = x
last (x:xs) = last xs
```

Transformação 1:

```
last xs = case xs of
            (x:[]) -> x
            (x:xs') -> last xs'
```

Transformação 2:

```
last xs = case xs of
            (x:xs') -> case xs' of
                         [] -> x
                         (x':xs'') -> last xs'
```

Records

- Generalização de tuplos: produtos cartesianos com campos etiquetados
- A ordem dos campos não é significativa

Exemplo em Haskell:

```
data Pessoa = P { nome::String, idade::Int }

> let myself = P {nome="Pedro",idade=36 }
> myself == P {idade=36,nome="Pedro"}
True
> nome myself
"Pedro"
> idade myself
36
```

Records: extensões à linguagem

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & | & \{\ell_1 = e_1; \ldots; \ell_n = e_n\} & \text{construção} \\
 & | & e.\ell & \text{selecção}
\end{array}
$$

Sintaxe análoga à de atributos e métodos de objectos.

Codificar *records* usando pares

$$\{x = 5; y = 7; z = 6\} \equiv \textbf{pair}\ 5\ (\textbf{pair}\ 7\ 6))$$
$$e.x \equiv \textbf{fst}\ e$$
$$e.y \equiv \textbf{fst}\ (\textbf{snd}\ e)$$
$$e.z \equiv \textbf{snd}\ (\textbf{snd}\ e)$$

- Associar cada etiqueta a uma posição fixa.
- Apenas possível quando as etiquetas são conhecidas estáticamente.
- Alternativa: extender a semântica operacional (ver a bibliografia).

Polimorfismo de *records*

A função

$$\lambda x.\ x.\text{weight}$$

pode ser aplicada a qualquer record com um campo *weight*. Exemplos:

1. $\{\text{size} = 10; \text{weight} = 100\}$
2. $\{\text{name} = \text{"}\textit{Mike}\text{"}; \text{weight} = 10\}$

- Este polimorfismo designa-se por polimorfismo de *records* ou polimorfismo de objectos.
- O Haskell não suporta este polimorfismo: os nomes de campos não são polimorficos.