Módulo III:

Expressividade do λ-calculus

MCC - 2ºano

José Bernardo Barros (jbb@di.uminho.pt)
José Carlos Bacelar Almeida (bacelar@di.uminho.pt)

Departamento de Informática
Universidade do Minho

Motivação

A analogia construída sobre o λ-calculus permite-nos identificar redução como computação. Mas como representar os *dados*?

Recordemos o que se pretende com a representação de um tipo de dados: *estabelecer uma correspondência entre os valores e operações desse tipo com λ-termos de tal forma que estes últimos se comportem como os originais.* Um exemplo:

$$[\![+]\!]\ [\![5]\!]\ [\![3]\!] \stackrel{\beta}{=} [\![5+3]\!]$$

i.e. a representação da operação de adição, quando aplicada às representações dos números 5 e 3, reduz-se na representação do número 8.

Em vez de nos concentrarmos sobre tipos de dados específicos vamos considerar *tipos de dados indutivos*, i.e. tipos de dados caracterizados pelo conjunto de *construtores* admissíveis[a].

Note-se que, mais do que a representação dos *valores* de um tipo de dados (o que é facilmente conseguido representando os construtores desse tipo), está em jogo a representação de um mecanismo que permita manipular a estrutura desses valores (o equivalente à *concordância de padrões* do *Haskell*). Vamos aqui rentabilizar a cultura dos **catamorfismos** já estudada em *Métodos de Programação I* — para cada tipo de dados vamos considerar o *catamorfismo* associado que nos permite decompor os valores desse tipo — o que designamos **iterador**.

[a] Tipos de dados indutivos são definidos em HASKELL pela declaração `data`.

Expressividade

Booleanos:

Comecemos por considerar um tipo particularmente simples: os *booleanos*. Este tipo dispõe unicamente de dois valores (`True` e False) e o *iterador* corresponde ao condicional que seleciona o resultado em função do argumento recebido.

```
data Bool = True | False
iterBool :: Bool -> a -> a -> a
iterBool True x _ = x
iterBool True _ x = x
```

(note que `iterBool` é simplesmente o *velhinho* `if-then-else-`).

COMENTÁRIO: `iterBool` deve ser identificado como sendo, na sua essência, o `cata` definido sobre o tipo `Bool` — precisamente da mesma forma que identificávamos o `foldr` como sendo basicamente o `cata` das listas. De facto, um nome alternativo para `iterBool` podia bem ser `foldrBool`[a].

[a] A única diferença é que o valor a ser destruído é aqui o primeiro argumento quando em `foldr` é o último.

Expressividade II

Vamos considerar a seguinte codificação...

$$\begin{aligned} \text{True} &\doteq \lambda xy.x \\ \text{False} &\doteq \lambda xy.y \\ \text{iterB} &\doteq \lambda bxy.b\ x\ y \end{aligned}$$

Podemos verificar que `iterB` satisfaz as regras da definição de `iterBool`. De facto, para quaisquer termos X_1, X_2, temos

$$\begin{aligned} \text{iterB True } X_1\ X_2 &\xrightarrow{\beta\,\twoheadrightarrow} X_1 \\ \text{iterB False } X_1\ X_2 &\xrightarrow{\beta\,\twoheadrightarrow} X_2 \end{aligned}$$

A codificação utilizada tem ainda uma propriedade que não deixa de ser interessante notar: o iterador acaba por ter um papel inócuo no resultado anterior — ele comporta-se essencialmente como a identidade[a]:

$$\begin{aligned} \text{iterB True} &\overset{\beta}{=} \text{True} \\ \text{iterB False} &\overset{\beta}{=} \text{False} \end{aligned}$$

O que nos indica que podemos utilizar o λ-termo associado a um dos valores como *os seus próprios destrutores.* O iterador passa assim a ser desnecessário (a identidade serve de iterador).

Exemplo: a função `or` poderia ser definida como,

```
or x y = iterBool x True y
```

Atendendo à propriedade dos valores funcionarem como seus próprios destrutores teríamos então

$$\lambda xy.x\ \text{True}\ y$$

[a] De facto, esta função é extensionalmente equivalente à identidade.

Expressividade III

Vimos, na codificação apresentada para o booleanos, que *os valores exibem a notável propriedade de encerrar a codificação do iterador*. Essa propriedades não é intrínseca à codificação utilizada mas antes um *reflexo de uma* ***técnica de codificação de qualquer tipo indutivo no λ-calculus***.

Para demonstrar que assim é procuraremos codificar outros tipos indutivos bem conhecidos.

Pares:

```
data Pair a b = MkPair a b (Pair a b)
iterPair :: (Pair a b) -> (a -> b -> c) -> c
iterPair (Pair x y) = \f->f x y
proj1 x = iterPair x (\x y->x)
proj2 x = iterPair x (\x y->y)
```

Concentremo-nos então no iterador (que já foi escrito de forma a enfatizar que pretendemos codificar o iterador juntamente com o valor a ser decomposto). A codificação é assim directa — um par de X_1, X_2 é representado como:

$$\lambda f.f\ X_1\ X_2$$

E, daqui, obtemos

$$\begin{array}{rcll} \text{Pair} & \doteq & \lambda xy.(\lambda f.f\ x\ y) & \equiv \quad \lambda xyf.f\ x\ y \\ \text{iterPair} & \doteq & \lambda p.p & (redundante) \\ \text{Proj1} & \doteq & \lambda p.p\ (\lambda xy.x) & \\ \text{Proj2} & \doteq & \lambda p.p\ (\lambda xy.y) & \end{array}$$

Expressividade IV — Naturais

```
data Nat = Zero | Succ Nat
iterNat :: Nat -> (a->a) -> a -> a
iterNat Zero = \f z-> z
iterNat (Succ n) = \f z -> f (iterNat n f z)
```

A representação dos naturais deverá ser então

$$\begin{aligned} \text{Zero} &\doteq \lambda f z.z \\ (\text{Succ } X_1) &\doteq \lambda f z.f\ (X_1\ f\ z) \end{aligned}$$

$$\ldots \textit{logo,} \qquad \text{Succ} \doteq \lambda n f z.f\ (n\ f\ z)$$

Exercício: Demonstre que a forma normal do natural n é dada pelo λ-termo $\lambda f z.f\ (f\ (f \cdots (f\ z)))$, onde ocorrem precisamente n iterações de f.

Com o iterador dos naturais estamos habilitados a construir um vasto conjunto de funções

```
isZero x = (iterNat x) (\x->False) True
plus x y = (iterNat x) Succ y
mult x y = (iterNat x) (plus y) Zero
```

que nos permite definir imediatamente

$$\begin{aligned} \text{IsZero} &\doteq \lambda x.x\ (\lambda r.\text{False})\ \text{True} \\ \text{Plus} &\doteq \lambda x y.x\ \text{Succ}\ y \\ \text{Mult} &\doteq \lambda x y.x\ (\text{Plus y})\ \text{Zero} \end{aligned}$$

Comentário: Existem funções simples sobre os naturais que são particularmente difíceis de codificar com o único recurso do *iterador*. Alguns exemplos são a função `pred` (predecessor) e o `factorial`. Esta dificuldade não é uma impossibilidade (a sua definição aparece adiante num resumo de λ-termos usuais) mas deixaremos os detalhes da sua definição para um outro módulo deste curso.

Recursividade genérica

Sabemos que nem todas as funções podem ser expressas como *catamorfismos* de tipos indutivos de dados. Coloca-se, por isso, a questão:

Como representar *funções recursivas* arbitrárias no λ-calculus?

Vamos considerar o exemplo paradigmático das funções recursivas... [a]

```
factorial 0 = 1
factorial (n+1) = (n+1) * (factorial n)
```

Para se compreender o significado de uma função recursiva é conveniente introduzir o seguinte conceito:

Seja $F : X \to X$ uma função. Um ***ponto fixo*** *de F é um elemento $x \in X$ tal que $F(x) = x$.* [b]

Neste ponto, basta-nos observar que podemos entender a função $factorial$ como um ponto fixo do operador $\mathcal{H}$ definido como:

$$\mathcal{H}(f)(n) = \begin{cases} 1 & \text{se } n = 0 \\ n * f(n-1) & \text{se } n > 0 \end{cases}$$

Este operador pode ser representado pelo λ-termo.

$$\lambda f n.(\text{IsZero } n)\ (\text{Succ Zero})\ (\text{Mult } n\ (f\ (\text{Pred } n)))$$

onde utilizamos as constantes definidas para os naturais.

[a] Como sabemos (MP1), a função $factorial$ não pode ser "directamente" expressa como um catamorfismo — mesmo se, tal como para o predecessor, é possível definir esta função com o iterador em conjugação com os pares (ver definição no *quadro de λ-termos)*..

[b] Existe um vasto conjunto de resultados matemáticos (desde *Tarski*, *D. Scott*, ...) que estabelece em que condições esse ponto fixo existe e como pode ser calculado. É, de facto, uma disciplina que, dado o papel proeminente da recursividade nas ciências da computação, ganhou estatuto próprio (e importantíssimo) e é designada por **Teoria de Domínios**. Mas, mais uma vez, temos de estabelecer limites para este curso remetendo esse estudo para outras disciplinas da licenciatura.

Operador de ponto fixo

O que é extraordinário é que todo o λ-termo tem um ponto fixo e esse ponto fixo pode ser calculado por um λ-termo.

Considere-se o termo

$$Y \quad \doteq \quad \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Para qualquer termo F, temos (verifique expandindo YF)

$$F\ (Y\ F) = Y\ F$$

Logo $Y\ F$ é um ponto fixo de F.

Voltando ao factorial...

Em resumo, vimos que podemos representar o factorial como o λ-termo

$$\text{Factorial} \quad \doteq \quad Y(\lambda f n.(\text{IsZero } n)\ (\text{Succ Zero})\ (\text{Mult } n\ (f\ (\text{Pred } n))))$$

Note-se que a técnica apresentada nos permite representar qualquer função recursiva (mesmo as que não terminam...). Por exemplo, a função

```
erro x = erro x
```

pode ser representada como o termo

$$YI \doteq Y(\lambda x.x) = (\lambda x.xx)\lambda x.xx \doteq \Omega$$

(verifique) que já vimos tratar-se de um termo que não dispõe de forma normal (a computação não termina).

Combinadores úteis

$$\begin{aligned}
\text{K} &\doteq \lambda xy.x \\
\text{S} &\doteq \lambda xyz.x\ z\ (y\ z) \\
\text{I} &\doteq \text{S K K} = \lambda x.x \\
\text{D} &\doteq \lambda x.x\ x \\
\Omega &\doteq DD \quad = \quad (\lambda x.x\ x)\lambda x.x\ x
\end{aligned}$$

booleanos

$$\begin{aligned}
\text{True} &\doteq K = \lambda xy.x \\
\text{False} &\doteq \lambda xy.y
\end{aligned}$$

pares

$$\begin{aligned}
\text{Pair} &\doteq \lambda xyz.x\ y \\
\text{Proj1} &\doteq \lambda p.p\ (\lambda xy.x) \\
\text{Proj2} &\doteq \lambda p.p\ (\lambda xy.y)
\end{aligned}$$

naturais

$$\begin{aligned}
\text{Zero} &\doteq \lambda zs.z \\
\text{Succ} &\doteq \lambda nzs.s\ (n\ z\ s) \\
\text{IsZero} &\doteq \lambda n.n\ (\lambda r.\text{False})\ \text{True} \\
\text{Plus} &\doteq \lambda xy.x\ \text{Succ}\ y \\
\text{Mult} &\doteq \lambda xy.x\ (\text{Plus}\ y)\ \text{Zero} \\
\text{PrimRecNat} &\doteq \lambda nfz.\text{Proj2}\ (n\ (\lambda p.\text{Pair}\ (\text{Succ}\ (\text{Proj1}\ p))\ (f\ (\text{Proj1}\ p)\ (\text{Proj2}\ p)))\ (\text{Pair Zero}\ z)) \\
\text{Pred} &\doteq \lambda n.\text{PrimRecNat}\ n\ (\lambda pr.p)\ \text{Zero} \\
\text{Factorial} &\doteq \lambda n.\text{PrimRecNat}\ n\ (\lambda pr.\text{Mult}\ (\text{Succ}\ p)\ r)\ (\text{Succ Zero})
\end{aligned}$$

op.p.fixo

$$\begin{aligned}
\text{Y} &\doteq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \\
\text{FactorialRec} &\doteq \text{Y}\ (\lambda fn.(\text{IsZero n})\ (\text{Succ Zero})\ (\text{Mult}\ n\ (f\ (\text{Pred}\ n))))
\end{aligned}$$