Ponto fixo

Definição

O "ponto fixo" de uma função f é o valor x tal que $f(x) = x$. Ou seja, corresponde ao objeto que não se modifica pela aplicação da função. Uma função pode ter um, mais de um ou nenhum ponto fixo.

- A função $f(x) = x^2$ possui como pontos fixos os valores 0 e 1, pois $0^2 = 0$ e $1^2 = 1$;
- A função $f(x) = x + 1$ não possui nenhum ponto fixo, pois $f(x) \neq x, \forall x \geq 0$;
- A função $f(x) = x * 2$ possui como ponto fixo apenas o valor 0, pois $0 * 2 = 0$.

Ponto fixo

Existência

No Cálculo Lambda, é possivel demonstrar que todo termo possui um ponto fixo. Ou seja, para todo termo X existe um termo P, dependente de X, tal que:

$$XP =_\beta P.$$

Além disso, esse ponto fixo pode ser obtido pela aplicação de um operador Y, de tal forma que, para todo termo X, YX é um ponto fixo de X.

Ponto fixo

Teorema

No Cálculo Lambda, existe um operador Y tal que, para todo termo x,

$$Yx =_\beta x(Yx) \quad \text{e, mais forte ainda,} \quad Yx \rhd_\beta x(Yx).$$

<u>Prova</u>:

- Basta considerar $Y \equiv UU$, com $U \equiv \lambda ux.x(uux)$.

Esse termo foi inventado por Alan Turing em 1937, e também é conhecido por Y_{Turing}.

Ponto fixo

Teorema

De fato:

$$
\begin{array}{rcl}
Yx & \equiv & (\lambda u.\lambda x.x(uux))Ux \\
 & \rhd_\beta & [U/u](\lambda x.x(uux))x \\
 & \equiv & (\lambda x.x(UUx))x \\
 & \rhd_\beta & x(UUx) \\
 & \equiv & x(Yx)
\end{array}
$$

Ponto fixo

Teorema

Y_{Turing} não é o único operador de ponto fixo conhecido. Exsitem outros, entre os quais $Y_{Curry-Ros}$, inventado por Haskell Curry:

$$Y_{Curry-Ros} \equiv \lambda x.VV, V \equiv \lambda y.x(yy)$$

Exercício: provar que $Y_{Curry-Ros}$ é um operador de ponto fixo, ou seja, que $YX =_{\beta} X(YX)$ para todo termo X.

Pontos fixos: combinador Y

Chamamos o termo abaixo de **combinador Y** (ou combinador de ponto fixo)

$$\mathbf{Y} = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Teorema: o combinador **Y** produz um ponto fixo para seu argumento

Demonstração:

$$\begin{array}{rr} \mathbf{Y}S & \rightarrow_\beta \\ (\lambda x.S(x\ x))\ (\lambda x.S(x\ x)) & \rightarrow_\beta \\ S\ (\lambda x.S(x\ x))\ (\lambda x.S(x\ x)) & =_\beta \\ S(\mathbf{Y}S) & \end{array}$$

$\mathbf{Y}S =_\beta S(\mathbf{Y}S)$ e, portanto, $\mathbf{Y}S$ é um ponto fixo de S □

Podemos finalmente definir $I = \mathbf{Y}S$.

Ponto fixo

Aplicação

A importância desse corolário reside no fato de que ele permite a resolução de formulações recursivas, sendo útil na construção de termos lambda que representam as funções correspondentes. É importante observar:

- No Cálculo Lambda as funções são anônimas, ou seja, elas não são identificadas;
- Dessa maneira, não é possível representar diretamente funções que invocam a si mesmas;
- No entanto, a partir da equação de define a função recursiva em questão, o uso operador de ponto fixo permite obter a expressão lambda que representa tal função.

Definições recursivas

Exemplo - fatorial

Exemplo clássico de definição recursiva:

$$fat(n) = \begin{cases} 1 \text{ se } n = 0; \\ n * fat(n-1) \text{ se } n > 0. \end{cases}$$

Essa equação pode ser escrita como:

$$xy =_\beta \overline{if}\ (\overline{zero}\ y)\ \overline{1}\ (\overline{mult}\ y\ x\ (\overline{sub}\ y\ \overline{1}))$$

Definições recursivas

Exemplo - fatorial

A solução em x é obtida considerando-se $Y(\lambda x y_1.Z)$, onde:

$$\begin{array}{rcl} y_1 & = & y \\ Z & = & \overline{if}\ (\overline{zero}\ y)\ \overline{1}\ (\overline{mult}\ y\ x\ (\overline{sub}\ y\ \overline{1})) \end{array}$$

ou seja:

$$\overline{fat} \equiv Y\underbrace{(\lambda xy.\underbrace{(\overline{if}\ (\overline{zero}\ y)\ \overline{1}\ (\overline{mult}\ y\ x\ (\overline{sub}\ y\ \overline{1})))}_{Z})}_{F} \equiv YF$$

Definições recursivas

Exemplo - fatorial

Exemplo: $fat(3)$

$$
\begin{aligned}
Y F\ \overline{3} & \ \rhd_\beta \\
F(Y F)\ \overline{3} & \ \rhd_\beta \\
(\lambda y.\ \overline{if}\ (\overline{zero}\ y)\ \overline{1}\ (\overline{mult}\ y\ ((Y F)(\overline{sub}\ y\ \overline{1}))))\ \overline{3} & \ \rhd_\beta \\
\overline{if}\ (\overline{zero}\ \overline{3})\ \overline{1}\ (\overline{mult}\ \overline{3}\ ((Y F)(\overline{sub}\ \overline{3}\ \overline{1}))) & \ \rhd_\beta \\
\overline{mult}\ \overline{3}\ ((Y F)\ \overline{2}) & \ \rhd_\beta \\
\overline{mult}\ \overline{3}\ (F(Y F)\ \overline{2}) & \ \rhd_\beta \\
\overline{mult}\ \overline{3}\ (\overline{mult}\ \overline{2}\ ((Y F)\ \overline{1})) & \ \rhd_\beta
\end{aligned}
$$

Definições recursivas

Exemplo - fatorial

continuação:

$$
\begin{array}{rl}
\overline{mult}\ \overline{3}\ (\overline{mult}\ \overline{2}\ (F(Y F)\ \overline{1})) & \triangleright_\beta \\
\overline{mult}\ \overline{3}\ (\overline{mult}\ \overline{2}\ (\overline{mult}\ \overline{1}\ ((Y F)\ \overline{0}))) & \triangleright_\beta \\
\overline{mult}\ \overline{3}\ (\overline{mult}\ \overline{2}\ (\overline{mult}\ \overline{1}\ (F(Y F)\ \overline{0}))) & \triangleright_\beta \\
\overline{mult}\ \overline{3}\ (\overline{mult}\ \overline{2}\ (\overline{mult}\ \overline{1}\ \overline{1})) & \triangleright_\beta \\
\overline{mult}\ \overline{3}\ (\overline{mult}\ \overline{2}\ \overline{1}) & \triangleright_\beta \\
\overline{mult}\ \overline{3}\ \overline{2} & \triangleright_\beta \\
\overline{6} &
\end{array}
$$

Definições recursivas

Exercício

Obter uma expressão lambda que calcula o n-ésimo termo da seqüência de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

$$\begin{cases} f(1) = 0; \\ f(2) = 1; \\ f(n) = f(n-1) + f(n-2) \text{ para } n \geq 3. \end{cases}$$

Pares ordenados

Definição: um par (M, N) pode ser representado por **pair** $M\ N$ onde

$$\textbf{pair} = \lambda\ m\ n\ b\ .\ b\ m\ n$$

Exemplo:

$$\textbf{pair 0 true} = \lambda b.\ b\ \textbf{0 true}$$

Definição: funções de acesso ao conteúdo de um par

$$\textbf{fst} = \lambda p.\ p\ \textbf{true}$$
$$\textbf{snd} = \lambda p.\ p\ \textbf{false}$$

Notação: usaremos $\langle M, N \rangle$ como um sinônimo para **pair** $M\ N$.

Listas

Listas podem ser consideradas as estruturas de dados fundamentais em linguagens de programação funcionais.

A definição algébrica de uma lista oferece dois construtores:

- `empty` (lista vazia)
- `cons`, que recebe um valor e uma lista, e retorna a lista prefixada pelo valor.

Exemplo:

```
empty
(cons 2 empty)
(cons 2 (cons 5 (cons 1 empty)))
```

Listas (2)

Três funções são definidas para acessar o conteúdo da lista:

- `isEmpty` testa se a lista é vazia
- `head` retorna o primeiro elemento da lista
- `tail` retorna a lista resultante da remoção do primeiro elemento.

Aplicar `head` ou `tail` sobre `empty` é um erro.

Exemplo:

```
isEmpty empty                      => true
isEmpty (cons 2 empty)             => false
head    (cons 2 empty)             => 2
tail    (cons 3 (cons 2 empty))    => (cons 2 empty)
head (tail (cons 3 (cons 2 empty))) => 2
```

Listas (3)

A seguinte codificação implementa as operações de listas sobre cálculo lambda.

Definição: construtores de listas:

$$\begin{aligned} \textbf{empty} &= \lambda x.\ \textbf{true} \\ \textbf{cons} &= \lambda h\ t.\ \textbf{pair}\ h\ t \end{aligned}$$

Definição: operações de lista:

$$\begin{aligned} \textbf{isEmpty} &= \lambda l.l\ (\lambda x\ \lambda y.\textbf{false}) \\ \textbf{head} &= \textbf{fst} \\ \textbf{tail} &= \textbf{snd} \end{aligned}$$

História

- Os primeiros resultados acerca da indecidbilidade em toda a história foram descobertos por Church através do Cálculo Lambda;
- Eles tratam da inexistência de algoritmos para determinar (i) se duas expressões lambdas satisfazem À relação $=_\beta$ (isto é, se elas representam a mesma operação) e (ii) determinar se uma expressão lambda possui forma normal ou não;
- A partir desses resultados, Church foi capaz de deduzir a indecidibilidade da lógica de predicados pura de primeira ordem em 1936.

História

- Com isso, ele respondeu uma questão formulada por David Hilbert anos antes (*Entscheidungsproblem*);
- *Entscheidungsproblem*: determinar a existência de um algoritmo que decide se uma certa afirmação pode ser provada a partir de axiomas usando as regras da lógica;
- A demonstração a seguir foi feita por Dana Scott em 1963 e redescoberta independentemente por Curry em 1972.

Universalidade de cálculo lambda

Universalidade

Cálculo-lambda como linguagem de programação é Turing-computável.

Há várias formas de provar isso, mas é fácil perceber que

- Usando duas listas podemos codificar uma fita bidirecional
- Podemos codificar estados e símbolos utilizando números
- Usando listas, pares, podemos codificar uma função de transição de estado
- Procedimentos recursivos pode ser utilizados para implementar consulta à função de transição de estado.

Com esses componentes, podemos codificar Máquinas de Turing.

Revisão

O que vimos:

- os componentes fundamentais da teoria de cálculo lambda.
- como utilizar o cálculo lambda puro como uma linguagem de programação funcional, utilizando diversos mecanismos de codificação.

O que não foi visto:

- a teoria de *lambda calculi* é vasta, e inclui diversas variações importantes com sistemas de tipos associados:
  - cálculo lambda simplesmente tipado
  - cálculo lambda polimórfico
  - cálculo lambda com tipos dependentes
- mesmo a teoria do cálculo lambda sem tipos possui diversos resultados que não foram citados. Para mais informações, ver as referências ao final.

Retomando a pergunta inicial ...

Suponha uma linguagem de programação com números e valores booleanos. Considere as seguintes construções:

1. execução condicional (`if-then` e `if-then-else`)
2. laços de repetição (`while` e `for`)
3. definição de variáveis e operador de atribuição
4. definição e aplicação de funções

Pergunta: se fosse necessário escolher *somente um* dos itens acima, ainda teríamos uma linguagem de programação expressiva o suficiente?

Resposta: certamente, desde que a escolha seja o item 4!

Referências

- The lambda calculus: its syntax and semantics (Barendregt, 1984)
- An introduction to functional programming through lambda calculus (Michaelson, 1989)
- Lectures on the Curry-Howard isomorphism (Sørensen and Urzyczyn, 2006)
- A short introduction to the Lambda Calculus (Jung, 2004) Disponível online em http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf
- How to Design Programs: An Introduction to Computing and Programming (Felleisen, Findler, Flatt, Krishnamurthi, 2003) Disponível online em http://www.htdp.org

Bibliografia

1. *Lambda-Calculus and Combinators - An Introduction*
   J. R. Hindley and J. P. Seldin
   Cambridge University Press, 2008
   Capítulos 1, 3, 4 e 5
2. *Teoria da Computação: Máquinas Universais e Computabilidade*
   T. A. Divério e P. B. Menezes
   Bookman, 2011, 3ª edição
   Capítulo 8
3. `http://en.wikipedia.org/wiki/Lambda_calculus`
4. `http://en.wikipedia.org/wiki/Church_encoding`
5. `http://ozark.hendrix.edu/~burch/proj/lambda/`