Lambda Cálculo

Programação Funcional

Prof. Maycon Amaro

Introdução

- O λ-cálculo é um modelo de computação desenvolvido em 1935 por Alonzo Church.
- Church era o orientador de doutorado de Alan Turing.
- O λ-cálculo é a base das linguagens funcionais. É um sistema muito simples e muito poderoso.
- É um sistema baseado em expressões.

Figure 1: Alonzo Church

Sintaxe

Possui apenas três construções:

- Variáveis: um nome para um potencial valor.
- Abstrações: definições de funções.
- Aplicações: a aplicação de uma função em seus argumentos.

Formalmente:

$$e := v \mid \lambda v.e \mid e\ e$$

Abstrações

- É uma definição de função. Funções no λ-cálculo não tem nome, e por isso são conhecidas como *funções anônimas*.
- Utilizamos a letra grega λ para marcar o início de uma abstração e uma letra minúscula para dar um nome ao parâmetro.
- Em seguida colocamos um ponto final para indicar o início do corpo da função, que pode inclusive ser outra abstração. Abstrações se estendem o máximo possível para a direita.

Exemplo: $\lambda x.x$ é uma função que simplesmente retorna seu parâmetro. Como a função `id` em Python abaixo:

```python
def id(x):
  return x
```

Variáveis livres e ligadas

Uma variável é ligada quando há um λ no seu escopo que a define como nome de parâmetro, e livre caso contrário.

No termo $\lambda x.x$, x é uma variável ligada.

No termo $\lambda x.y$, y é uma variável livre.

Um termo sem variáveis livres é chamado de **termo fechado**.

Variáveis livre e ligadas

Numa linguagem de programação, uma variável livre pode ser pensada como o nome de um valor ou função da biblioteca padrão. Podemos fazer uma analogia com o `pi` em Python:

```
def f(x):
  return math.pi
```

α-equivalência

Variáveis ligadas podem ser renomeadas sem alterar o significado da abstração.

O termo $\lambda x.x$ é α-equivalente ao termo $\lambda y.y$.

Aplicações

Uma aplicação envolve duas expressões. Se a expressão da esquerda é uma abstração, isso é chamado de **redex** e pode ser posteriormente *reduzido* ou *avaliado*. Senão, nada pode ser feito.

O termo $(\lambda x.x)\, y$ é uma aplicação da abstração $\lambda x.x$ sobre a variável livre y. Isso pode ser reduzido.

O termo $x\, y$ é uma aplicação de uma variável livre sobre outra. Não há como reduzir.

Aplicações

Como abstrações se estendem ao máximo para direita, parênteses são necessários. O termo $\lambda x.x\ y$ é simplesmente uma abstração, cujo corpo aplica a variável ligada x à variável livre y.

β-redução

O lambda cálculo possui uma *semântica* bem definida, que nos permite reduzir termos. Informalmente, substituímos a variável ligada pelo termo que foi aplicado e eliminamos o λ.

Exemplo: O termo $(\lambda x.x)\ y$ pode ser reduzido para apenas y.

Capture-avoiding substitution

É possível que os nomes das variáveis envolvidas numa aplicação causem conflito.

O termo $(\lambda x.\lambda y.x\ y)\ y$ seria reduzido para $\lambda y.y\ y$. Uma variável que era livre se tornou ligada.

Para resolver isso, usamos a α-equivalência para renomear variáveis ligadas e eliminar o conflito.

Assim, $(\lambda x.\lambda y.x\ y)\ y$ é equivalente $(\lambda x.\lambda z.x\ z)\ y$ que seria reduzido para $\lambda z.y\ z$.

Essa substituição que renomeia variáveis ligadas conforme necessário é chamada de *capture-avoiding substitution*.

β-redução

Com isso, uma possível definição da β-redução é:

$$(\lambda v.e_1)\, e_2 \longrightarrow [v \mapsto e_2]e_1$$

em que $[v \mapsto e_2]e_1$ significa *capture-avoiding substitution* da "variável ligada" v pelo termo e_2 na expressão e_1.

$$\begin{aligned} (\lambda x.x)\, y \\ [x \mapsto y]x \\ y \end{aligned}$$

Outro exemplo

$$
\begin{aligned}
(\lambda k.\lambda c.k\ c)\,(\lambda b.b\ c) \\
[k \mapsto \lambda b.b\ c]\lambda c.k\ c \\
\lambda d.(\lambda b.b\ c)\ d
\end{aligned}
$$

Dependendo da *estratégia de avaliação*, esse termo poderia ser ainda mais reduzido.

Shadowing

Como abstrações se estendem ao máximo para a direita, nomes idênticos de parâmetros fazem com estejam ligados à abstração mais interna.

No termo $\lambda x.\lambda x.x\,x$, as duas ocorrências de x no corpo se referem ao λ mais interno.

No termo $\lambda x.(\lambda x.x\,x)\,x$ a qual λ pertence cada x?

Em algumas linguagens de programação, é permitido que variáveis num escopo mais interno tenham o mesmo nome de uma em um escopo mais externo, e isso é conhecido como **shadowing**.

β-redução

Capture-avoiding substitution substitui apenas as ocorrências do parâmetro que se tornam livres após a remoção do λ.

$$\begin{aligned} (\lambda x.(\lambda x.x\;x)\;x)\;z \\ [x \mapsto z]((\lambda x.x\;x)\;x) \\ (\lambda x.x\;x)\;z \end{aligned}$$

Isso ainda pode ser reduzido mais uma vez, resultando em $z\;z$.

β-redução

Com isso, uma possível definição formal da β-redução é:

$$(\lambda v.e_1)\, e_2 \longrightarrow [v \mapsto e_2]e_1$$

em que $[v \mapsto e_2]e_1$ significa *capture-avoiding substitution* das ocorrências livres de v em e_1 pelo termo e_2.

Forma Normal

Quando um termo não pode mais ser reduzido, ele se encontra na *forma normal*.

O termo $\lambda x.x$ é uma forma normal.

O termo $z\ y$ é uma forma normal.

Nem todo termo possui uma forma normal. O que acontece se tentarmos reduzir o termo $(\lambda x.x\ x)\ (\lambda x.x\ x)$? Este é um exemplo de termo que **diverge**.

Estratégias de Avaliação

Em qual ordem realizar as reduções?

$$(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z))$$

Full beta-reduction

Nessa estratégia, qualquer redex pode ser reduzido a qualquer momento. Uma possível redução até a *forma normal*:

$$
\begin{aligned}
(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z)) \\
(\lambda x.x)\,((\lambda x.x)\,(\lambda z.([x \mapsto z]x))) \\
(\lambda x.x)\,((\lambda x.x)\,(\lambda z.z)) \\
[x \mapsto ((\lambda x.x)\,(\lambda z.z))]x \\
(\lambda x.x)\,(\lambda z.z) \\
[x \mapsto \lambda z.z]x \\
\lambda z.z
\end{aligned}
$$

Não importa a ordem escolhida, a forma normal (quando existir) será a mesma. Essa propriedade é chamada de **confluência**.

Normal Order

Nessa estratégia, os redexes mais externos e mais à esquerda são reduzidos primeiro.

$$
\begin{aligned}
&(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z))\\
&[x \mapsto ((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z))]x\\
&(\lambda x.x)\,(\lambda z.(\lambda x.x)\,z)\\
&[x \mapsto \lambda z.(\lambda x.x)\,z]x\\
&\lambda z.(\lambda x.x)\,z\\
&\lambda z.[x \mapsto z]x\\
&\lambda z.z
\end{aligned}
$$

Call by Name

Essa estratégia é uma normal order em que não é permitido reduzir redexes que estão dentro de uma abstração.

$$
\begin{aligned}
(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z))\\
[x \mapsto ((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z))]x\\
(\lambda x.x)\,(\lambda z.(\lambda x.x)\,z)\\
[x \mapsto \lambda z.(\lambda x.x)\,z]x\\
\lambda z.(\lambda x.x)\,z
\end{aligned}
$$

Call by Value

Nessa estratégia, os redexes mais externos são reduzidos primeiro e somente depois da expressão à direita se tornar um *valor* (neste caso, uma forma normal).

$$\begin{aligned}
(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z)) \\
(\lambda x.x)\,([x \mapsto \lambda z.(\lambda x.x)\,z]x) \\
(\lambda x.x)\,(\lambda z.(\lambda x.x)\,z) \\
[x \mapsto \lambda z.(\lambda x.x)\,z]x \\
\lambda z.(\lambda x.x)\,z
\end{aligned}$$

Call by name *vs* Call by value

O termo $(\lambda x.z)((\lambda x.x)\ y)$ é avaliado para z independente da estratégia.

No *call by value*, o termo da direita será avaliado mesmo não sendo necessário.

Já no *call by name*, o termo da direita acaba sendo descartado sem ser avaliado.

Preguiçoso *vs* Estrito

Chamamos a estratégia *call by name* de **avaliação preguiçosa** ou **lazy evaluation**. Haskell utiliza uma variante dessa estratégia, chamada de *call by need*.

A maioria das linguagens utiliza a call by value, que é conhecida como **avaliação estrita** ou **eager evaluation**.

Como programar usando λ-cálculo?

Números naturais

Na *notação de Peano*, os números naturais são escritos de maneira recursiva.

- 0 é um número natural.
- O sucessor de um número natural é um número natural.

Assim, 1 pode ser escrito como S0 (sucessor de zero). 2 pode ser escrito como SS0 (sucessor do sucessor de 0).

Church numerals

Números no λ-cálculo são duas abstrações aninhadas, que aplica o primeiro parâmetro no segundo n vezes para representar o número n.

- 0 é $\lambda s.\lambda z.z$
- 1 é $\lambda s.\lambda z.s\ z$
- 2 é $\lambda s.\lambda z.s\ (s\ z)$
- 3 é $\lambda s.\lambda z.s\ (s\ (s\ z))$

Função Sucessor

Podemos criar um termo que calcula o sucessor de um *Church numeral*.

$$\lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$$

Exemplo, calculando o sucessor de 1 (com *full-beta reduction*):

$$\begin{aligned}
(\lambda n.\lambda s.\lambda z.s\ (n\ s\ z))\ (\lambda s.\lambda z.s\ z) \\
\lambda s.\lambda z.s\ ((\lambda s.\lambda z.s\ z)\ s\ z) \\
\lambda s.\lambda z.s\ ((\lambda z.s\ z)\ z) \\
\lambda s.\lambda z.s\ (s\ z)
\end{aligned}$$

A forma normal alcançada é o *church numeral* que representa 2.

Função Soma

Termo que calcula a soma de dois *church numerals*

$$\lambda n.\lambda m.\lambda s.\lambda z.n\, s\, (m\, s\, z)$$

Exemplo: $1 + 2$

$$\begin{aligned}
(\lambda n.\lambda m.\lambda s.\lambda z.n\, s\, (m\, s\, z))\, (\lambda s.\lambda z.s\, z)\, (\lambda s.\lambda z.s\, (s\, z)) \\
(\lambda m.\lambda s.\lambda z.(\lambda s.\lambda z.s\, z)\, s\, (m\, s\, z))\, (\lambda s.\lambda z.s\, (s\, z)) \\
\lambda s.\lambda z.(\lambda s.\lambda z.s\, z)\, s\, ((\lambda s.\lambda z.s\, (s\, z))\, s\, z)) \\
\lambda s.\lambda z.(\lambda z.s\, z)\, (s\, (s\, z))) \\
\lambda s.\lambda z.s\, (s\, (s\, z)))
\end{aligned}$$

A forma normal alcançada é a representação de 3.

Exercício

Escreva um termo que realize a multiplicação de dois *church numerals* e mostre a execução de 2×3.

Dicas e detalhes do exercício estarão no Moodle.

Church Booleans

O falso é representado por $\lambda t.\lambda f.f$, que recebe dois argumentos e retorna o segundo. Note que isso é α-equivalente à 0.

O verdadeiro é representado por $\lambda t.\lambda f.t$, que recebe dois argumentos e retorna o primeiro.

Um `if-then-else` é então um termo que aplica o booleano recebido nos outros dois parâmetros:

$$\lambda b.\lambda m.\lambda n.b\ m\ n$$

Exemplo

O seguinte termo testa se um church numeral é 0. Aqui, f representa o termo para falso e t o termo para verdadeiro.

$$\lambda n.n\,(\lambda x.f)\;t$$

Aplicando esse termo em 1, teremos o termo que representa o falso:

$$\begin{aligned}
(\lambda n.n\,(\lambda x.f)\;t)\,(\lambda s.\lambda z.s\;z)\\
(\lambda s.\lambda z.s\;z)\,(\lambda x.f)\;t\\
(\lambda z.(\lambda x.f)\;z)\;t\\
(\lambda x.f)\;t\\
f
\end{aligned}$$

Exemplo

Aplicando esse termo em 0, teremos o termo que representa o verdadeiro:

$$\begin{aligned}
(\lambda n.n\,(\lambda x.f)\;t)\,(\lambda s.\lambda z.z) \\
(\lambda s.\lambda z.z)\,(\lambda x.f)\;t \\
(\lambda z.z)\;t \\
t
\end{aligned}$$

Exemplo

Um termo para `if y == 0 then 1 else 2`. Aqui, f representa o falso, t representa o verdadeiro, e $c1$ e $c2$ são os termos que representam os números 1 e 2.

$$\lambda y.(\lambda b.\lambda m.\lambda n.b\ m\ n)\,((\lambda n.n\,(\lambda x.f\,)\ t)\ y)\ c1\ c2$$

Exercícios

1. Escreva a redução da aplicação deste termo com a representação de 3 e constate que o resultado será a representação de 2.
2. Escreva um termo que realize a conjunção (`and`) de dois *church booleans*.

Listas

A notação recursiva de uma lista é

- `[]` é uma lista
- Se `xs` é uma lista, então um elemento adicionado a `xs` é uma lista.

A lista `[1]` é `1 :: []`, e a lista `[2, 1, 6]` é `2 :: (1 :: (6 :: []))`.

Listas e Tuplas em λ-cálculo

O termo para $\lambda a.\lambda b.\lambda f.f\ a\ b$ pode ser usado para construir uma tupla. Assim, o termo $\lambda f.f\ a\ b$ representa `(a, b)`.

O termo $\lambda x.\lambda r.\lambda f.r$ representa `[]`. Note que nele há um subtermo α-equivalente ao verdadeiro.

Podemos representar `x::xs` como `(x, xs)`, ou seja, podemos usar tuplas aninhadas para construir uma lista.

Usando c_i para representar o *church numeral* i, a lista `[2, 1, 6]` é:

$$\lambda f.f\ c_2\ (\lambda f.f\ c_1\ (\lambda f.f\ c_6\ (\lambda x.\lambda r.\lambda f.r)))$$

Exercício

Escreva um termo que verifique se uma lista é vazia.

Recursão (Operador de ponto-fixo)

O seguinte termo é chamado de *call by value Y-combinator*. Utilizando a estratégia *call by value*, o termo que lhe for passado será repetido indefinidamente.

$$\lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\,(\lambda x.f\ (\lambda y.x\ x\ y))$$

Recursão (Operador de ponto-fixo)

Há uma versão mais simples, mas que só funciona em *call by name*, já que *call by value* tentaria reduzir o termo da direita, causando um *loop* infinito.

$$\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

Turing Completeness

- O λ-cálculo pode ser usado para escrever e computar qualquer função computável. Dizemos que ele é *turing-completo*, pois é equivalente às máquinas de Turing.
- No entanto, a situação muda um pouco se adicionarmos tipos ao sistema.
- Um bom interpretador pode ser utilizado em https://lambster.dev. Pode ajudar com os exercícios!

A influência do λ-cálculo

Funções anônimas

O seguinte programa em Python imprime 3.

```python
(lambda x : x + 1)(2)
```

A função abaixo em C++11 calcula o sucessor de um número.

```cpp
[](int x){ return x + 1; }
```

Arrow functions em JavaScript são funções anônimas.

```javascript
const hi = () => { console.log("hi") }
```

Closures

Um termo com uma variável livre pode se tornar fechado se o transformamos em uma aplicação.

Suponha que há uma variável z no escopo. Um **closure** de $\lambda x.x + z$ se tornaria $(\lambda y.\lambda x.x + y)\, z$.

Capturamos a variável e a transformamos em um parâmetro. Assim, esse termo continuaria tendo o mesmo significado sem depender de variáveis livres na abstração.

As funções anônimas implementadas em linguagens modernas costumam fazer isso automaticamente.

Shadowing

Em Rust, isso imprime 4 e depois 5.

```rust
let x = 5;
{ let x = 4;
  println!("{x}");
}
  println!("{x}");
```

Em JavaScript, isso imprime 4 e depois 5.

```javascript
let a = 4;
if (true) {
  let a = 5;
  console.log(a);
}
console.log(a);
```

Veremos mais!

Conforme avançamos na disciplina, veremos como conceitos de programação funcional, que se originam no λ-cálculo, influenciam diversas linguagens.