Uma Implementação do Cálculo Lambda não Tipado em Elixir

Christian S. Lima[1], **Adolfo Neto**[1]

[1] Universidade Tecnológica Federal do Paraná (UTFPR)
Curitiba, Brasil

christiansantoslima21@gmail.com, adolfo@utfpr.edu.br

Abstract. *In this paper we realized a bibliographic revision about untyped lambda calculus and developed an interpreter in Elixir.*

Resumo. *Neste artigo realizamos uma revisão bibliográfica sobre o cálculo lambda não tipado e construímos um interpretador em Elixir.*

1. Introdução

O cálculo lambda foi criado por Alonzo Church em [Church 1932]. A ideia inicial era fundamentar a lógica e a matemática. Futuramente viria a ser a base para linguagens funcionais.

Na seção 2, apresentamos o cálculo lambda não tipado, a noção de forma normal e como defini-las usando β-reduções. Consultamos alguns dos principais livros de referência sobre o cálculo lambda para a realização da revisão bibliográfica deste artigo, a saber: [Hindley and Seldin 2008], [Hindley 1997], [Sørensen and Urzyczyn 2006] e [Barendregt 1984]. Também consultamos alguns textos mais didáticos online, a saber: [Brandl a] e [Brandl b].

Na seção 3, descrevemos nosso interpretador do cálculo lambda não tipado. Para implementá-lo, utilizamos técnicas apresentadas nos livros [Nystrom 2021] e [Aho and Lam 2022]. O código fonte está disponível em: `https://github.com/Every2/An-implementation-of-untyped-lambda-calculus`.

2. Cálculo Lambda não Tipado

2.1. Linguagem

Definição 1 O alfabeto do cálculo lambda é dado pelos seguintes símbolos:

- um conjunto de variáveis:

$$\mathrm{Var} = \{x_i : i \in \mathbb{N}\};$$

- um abstrator: λ;
- três delimitadores: "(", ".", ")".

Definição 2 Os λ-termos são definidos de forma indutiva pelas regras:

1. todas as variáveis são λ-termos;
2. se M e N são λ-termos, então (MN) é um λ-termo (chamado de aplicação);

3. Se M é um λ-termo e x uma variável, então $(\lambda x.M)$ é um λ-termo (chamado abstração).

Definição 3 Definimos recursivamente o conjunto das variáveis que ocorrem livres em um λ-termo M pelas regras:

1. $\mathrm{FV}[x] = \{x\}$;
2. $\mathrm{FV}[NP] = \mathrm{FV}[N] \cup \mathrm{FV}[P]$;
3. $\mathrm{FV}[\lambda x.N] = \mathrm{FV}[N] - \{x\}$.

Definição 4 Definimos recursivamente a substituição de todas as ocorrências livres de x por N pelas regras:

1. $x[x := N] = N$;
2. $y[x := N] = y$, se $x \neq y$;
3. $(PQ)[x := N] = P[x := N]Q[x := N]$;
4. $(\lambda x.P)[x := N] = \lambda x.P$;
5. $(\lambda y.P)[x := N] = \lambda y.P$ se $x \notin \mathrm{FV}[P]$;
6. $(\lambda y.P)[x := N] = \lambda y.P[x := N]$ se $x \in \mathrm{FV}[P]$ e $y \notin \mathrm{FV}[N]$;
7. $(\lambda y.P)[x := N] = \lambda z.P[y := z][x := N]$ se $x \in \mathrm{FV}[P]$ e $y \in \mathrm{FV}[N]$.

Definição 5 **(α-conversão)** Seja um termo P e que contém uma abstração $\lambda x.M$ como subtermo e seja $y \notin \mathrm{FV}[M]$. Uma α-conversão de P é um termo Q obtido a partir de P substituindo uma ou mais ocorrências do subtermo $\lambda x.M$ por $\lambda y.M[x := y]$. Se Q é uma α-conversão de P, escrevemos $P \equiv_\alpha Q$.

2.2. β-redução

Definição 6 Um redex é um termo da forma $(\lambda x.M)N$.

Definição 7 Seja um termo P e que contém um redex da forma $(\lambda x.M)N$. Uma β-contração de P é um termo Q obtida a partir de P substituindo uma ocorrência do redex $(\lambda x.M)N$ por $M[x := N]$. Denotamos essa relação por $P \rightarrow_{1\beta} Q$.

Definição 8 Seja um termo P. Uma β-redução de P é um termo Q obtido a partir de P por uma sequência da forma:

$$P \equiv_\alpha P' \rightarrow_{1\beta} P_1 \equiv_\alpha P_1' \rightarrow_{1\beta} \ldots \rightarrow_{1\beta} P_n \equiv_\alpha Q$$

Quando Q é uma β-redução de P, escrevemos $P \rightarrow_\beta Q$.

Definição 9 Dizemos que um termo P está na forma normal quando nenhum dos seus subtermos é um redex. Quando $P \rightarrow_\beta Q$ e Q é uma forma normal, dizemos que Q é uma forma normal de P.

Teorema 10 *(Teorema de Church-Rosser) Se $P \rightarrow_\beta M$ e $P \rightarrow_\beta N$, então existe um termo Q tal que $M \rightarrow_\beta Q$ e $N \rightarrow_\beta Q$.*

Corolário 11 *A forma normal de um termo P, se existe, é única, a menos de α-conversão.*

A provas dos resultados acima pode ser encontrada em [Hindley and Seldin 2008].

Nem todo termo pode ser reduzido a uma forma normal, como vemos no exemplo abaixo:

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &\rightarrow_{1\beta} (\lambda x.xx)(\lambda x.xx)\\ &\rightarrow_{1\beta} (\lambda x.xx)(\lambda x.xx)\\ &\rightarrow_{1\beta} \ldots\end{aligned}$$

3. O Interpretador

O interpretador do Cálculo Lambda funciona como um REPL (Read-Eval-Print Loop), ou seja, ele roda como uma função recursiva que roda infinitamente e avalia as expressões. Ele é dividido em 3 etapas, sendo elas:

3.1. Lexer

O lexer é a primeira etapa do interpretador, onde recebemos uma string e a convertemos em uma sequência de tokens.

Um token tem a seguinte estrutura: um tipo que o identifica e um lexema, um símbolo que representa aquele token. Os tokens podem ser dos tipos seguintes:

- Variable (variável);
- Lambda (o símbolo de abstração, λ);
- LeftParen (parênteses esquerdo);
- RightParen (parênteses direito);
- Dot (ponto).

A função responsável por converter um caractere em um token é a função new, ela recebe um caractere e retorna uma das opções disponíveis acima, caso contrário retorna nil.

Para retornarmos uma sequência de tokens temos a função tokenizer. Ela recebe uma string, retira os espaços em branco com String.replace(), divide em grafemas (para evitar problemas com caracteres serem convertidos para sua forma em ASCII) e percorre a lista de grafemas com Enum.reduce_while, transformando cada grafema em um token, e retornando uma lista de tokens. Caso algum caractere não possa ser convertido em um token, retorna nil. No final, a lista de tokens é invertida com Enum.reverse para preservar a ordem original.

Listing 1. Função `tokenizer`

```
  def tokenizer(expr) do
   expr
   |> String.replace("■", "")
   |> String.graphemes()
   |> Enum.reduce_while([], fn c, acc ->
     case Tokens.new(c) do
       nil -> {:halt, nil}
       token -> {:cont, [token | acc]}
     end
   end)
   |> case do
     nil -> nil
     tokens -> Enum.reverse(tokens)
   end
 end
```

3.2. Parser

A etapa de parsing é a segunda etapa do interpretador. Nela fazemos a análise da sequência de tokens gerada pelo lexer e verificamos se corresponde à gramática do Cálculo Lambda, gerando a árvore sintatica do termo.

A lista de tokens pode se tornar o seguinte:

- Variable (variável): uma struct com apenas um campo chamado `name`;
- Abstraction (abstração): uma struct com dois campos, sendo o primeiro `param` (uma variável) e o segundo `term` (um termo);
- Application (aplicação): uma struct com dois campos, sendo o primeiro `lterm` (termo da esquerda) e o segundo `rterm` (termo da direita).

Uma varíavel como dito tem uma estrutura parecida com a do Lexer, mas dessa vez armazenamos apenas o campo `lexeme` em `name` como descrito na estrutura acima

Listing 2. Função `parse var`

```
def parse_var(tvec) do
  if length(tvec) == 1 do
    elem = Enum.at(tvec, 0)

    if elem.type == :variable do
      %Variable{name: elem.lexeme}
    else
      nil
    end
  else
    nil
  end
end
```

Já uma abstração é avaliada do seguinte modo: se o tamanho for menor que 4 e o primeiro token não for um parênteses esquerdo e o último não for um parênteses direto, retorna nil. Caso contrário, passamos para próxima fase onde verificamos se o segundo token é lambda e o quarto um ponto. Caso for verdade, verificamos se o tipo do terceiro token é uma variável e analisamos como tal, se não retornamos nil. Em caso de sucesso olhamos da quinta posição até a penúltima e analisamos recursivamente. Assim, criamos uma abstração com um parâmetro e um termo, caso contrário retornamos nil.

Listing 3. Função `parse abstraction`

```
def parse_abstraction(tvec) do
  cond do
    length(tvec) < 4 ->
      nil

    Enum.at(tvec, 0).type != :left_paren or List.last(tvec).type != :right_paren ->
      nil

```

```
      true ->
        if Enum.at(tvec, 1).type == :lambda and Enum.at(tvec, 3).type == :dot do
          lnode =
            case Enum.at(tvec, 2).type do
              :variable -> %Variable{name: Enum.at(tvec, 2).lexeme}
              _ -> nil
            end

          slice = 4..(length(tvec) - 2) |> Enum.map(fn x -> Enum.at(tvec, x) end)
          rnode = parse_tokens(slice)

          if rnode != nil do
            %Abstraction{param: lnode, term: rnode}
          else
            nil
          end
        else
          nil
        end
    end
  end
```

Seguimos a mesma estrutura de verificação para uma aplicação. Fazemos um slice da segunda posição até a última para contarmos o número de parênteses. Percorremos até encontramos o mesmo número de parênteses à esquerda e à direita, encerrando o processo com halt. Então analisamos a expressão de dentro dos parênteses e adicionamos no `lterm` e a da direita no `rterm`, assim formando uma aplicação.

Listing 4. Função `parse application`

```
  def parse_application(tvec) do
    cond do
      Enum.at(tvec, 0).type != :left_paren or List.last(tvec).type != :right_paren ->
        nil

      true ->
        slice = 1..(length(tvec) - 1) |> Enum.map(fn x -> Enum.at(tvec, x) end)


        {pos, _, _} =
          Enum.reduce_while(slice, {1, 0, 0}, fn x, {i, lp, rp} ->
            new_lp =
```

```
            if x.type == :left_paren do
              lp + 1
            else
              lp
            end

          new_rp =
            if x.type == :right_paren do
              rp + 1
            else
              rp
            end

          if new_lp == new_rp do
            {:halt, {i + 1, new_lp, new_rp}}
          else
            {:cont, {i + 1, new_lp, new_rp}}
          end
        end)

      lslice = 1..(pos - 1) |> Enum.map(fn x -> Enum.at(tvec, x) end)
      lnode = parse_tokens(lslice)
      rslice = pos..(length(tvec) - 2) |> Enum.map(fn x -> Enum.at(tvec, x) end)
      rnode = parse_tokens(rslice)

      if lnode != nil and rnode != nil do
        %Application{lterm: lnode, rterm: rnode}
      else
        nil
      end
    end
  end
```

3.3. Operações

A última etapa é executar as operações de α-conversão e β-redução na árvore sintática que construímos.

Dessa forma, precisamos verificar se as variáveis são livres em um termo, no caso, se todas as suas ocorrências não estão ligadas a uma abstração no termo. A função `is_free` percorre recursivamente o termo e verifica se a variável ocorre fora de alguma abstração e, portanto, é livre.

Definimos funções para realizarem operações definidas anteriormente:

- `replace`: executa a substituição de uma variável por um termo;
- `alpha`: executa a α-conversão de uma abstração, trocando uma variável por outra, caso seja possível;

- `beta`: executa a β-contração de uma aplicação, caso seja possível;
- `is_redex`: verifica se o termo é um redex.

Para executar a substituição, em um dos casos contamos com a função auxiliar `get_var_for_alpha`. Essa função auxilia a encontrar uma variável nova apropriada para executar uma α-conversão.

A função `exec` é encarregada de executar todas as β-reduções e reduzir o termo até sua forma normal.

Listing 5. Função `exec`

```
def exec(term) do
  case term do
    %Variable{name: _} ->
      term

    %Abstraction{param: x, term: rterm} ->
      new_rterm = exec(rterm)
      %Abstraction{param: x, term: new_rterm}

    %Application{lterm: lterm, rterm: rterm} ->
      if is_redex(term) do
        beta(term) |> exec()
      else
        new_lterm = exec(lterm)
        new_rterm = exec(rterm)
        new_term = %Application{lterm: new_lterm, rterm: new_rterm}

        if is_redex(new_term) do
          exec(new_term)
        else
          new_term
        end
      end
  end
end
```

Referências

Aho, A. and Lam, M. S. (2022). *Compilers principles techniques & tools*. Pearson.

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*. Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., New York, N.Y.

Brandl, H. Lambda calculus - step by step. `https://hbr.github.io/Lambda-Calculus/lambda1/untyped_lambda.pdf`. Acessado em: 14/02/2025.

Brandl, H. Programming with lambda calculus. `https://hbr.github.io/Lambda-Calculus/lambda2/lambda.html`. Acessado em: 14/02/2025.

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366.

Hindley, J. R. (1997). *Basic simple type theory*. Number 42. Cambridge University Press.

Hindley, J. R. and Seldin, J. P. (2008). *Lambda-calculus and combinators: an introduction*. Cambridge University Press.

Nystrom, R. (2021). *Crafting interpreters*. Genever Benning.

Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard isomorphism*. Elsevier.