Linguagens de Programação

Trabalho Prático 1

May 7, 2017

Discentes:

Hiago Oliveira - 29248
Gil Catarino - 32378

Docente:

Teresa Gonçalves

Introdução

Este trabalho foi feito com o objectivo de criar um avaliador de cálculo lambda. Nesta primeira fase ainda não avalia, apenas faz as α-equivalencias dos termos. Foi desenvolvido recorrendo às ferramentas **flex** e **bison** para facilitar no parsing e construção das árvores de sintaxe abstratas.

Funcionamento

Tal como referido na introdução, o interpretador começa por fazer uma análise lexical do input, transformando-o em tokens (flex) para de seguida ser feita a análise sintática de acordo com a gramática independente do contexto pretendida. A gramática BNF do cálculo lambda é dada por

- <termo> ::= <variavel>
- | (!<variavel>.<termo>)
- | (<termo> <termo>)

Contudo nesta gramática existe um conflito de mudança/redução devido à última regra da gramática. De forma a resolver este problema, foi adicionada outra regra, que elimina o problema e garante que as aplicações são associem à esquerda.

- <termlist> ::= <termlist> <termo>
- | <termo>
- <termo> ::= <variavel>
- | (!<variavel>.<termlist>)

Está quase, porem ainda temos de arranjar uma forma para fazer com que as nossas λ-abstrações sejam estendidas o mais à direita possível, coisa que com esta gramática não o faz. Foram introduzidas duas novas regras com vista a resolver este problema.

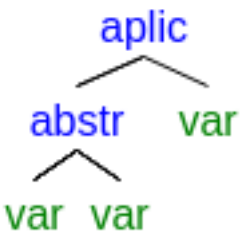
- <termlist> ::= <termlist> <termo>
- | <termo>
- <termo> ::= <variavel>
- | (<lamlist> <termlist>)
- <lamlist> ::= <lamlist> <lam>
- | <lam>
- <lam> ::= !<variavel>.

Árvores de Sintaxe

No seguimento da análise sintática é criada a árvore de sintaxe abstrata de acordo com a gramática imposta. A árvore conta com 3 tipos de nós: variável, aplicação (@), λ-abstração (λ). Foram portanto estes 3 tipos de nós que foram implementados no trabalho.

```
7   struct node_ {
8       enum {
9           var,
10          abstr,
11          aplic
12      } kind;
13      union {
14          char *v;
15          struct {
16              node n1;
17              node n2;
18          } nodestruct;
19      } u;
20  };
```

Um exemplo da árvore gerada pelo programa para o termo (λx.x) y



Aplicando a α-equivalência

Para aplicar a α-equivalência é necessário primeiro descobrir quais são as variáveis livres, pois estas não podem ser renomeadas, apenas as ligadas e as ligadoras.

- As variáveis livres podem ser descobertas através da seguinte forma:
- Nó de variavél - essa mesma variável
- Nó de abstração - variáveis livres do termo da abstração - variável do λ
- Nó de aplicação - União das variáveis livres das duas partes.

O algoritmo transformado em código fica então assim:

```
23  static listnode *varslivres(node *r)
24  {
25      listnode *vars;
26      listnode *temp1;
27      listnode *temp2;
28      switch ((*r)->kind) {
29          case var:
30              vars = new_list();
31              add(&vars, (*r)->u.v);
32              break;
33          case aplic:
34              vars = new_list();
35              temp1 = varslivres(&(*r)->u.nodestruct.n1);
36              temp2 = varslivres(&(*r)->u.nodestruct.n2);
37              join(&vars, temp1, temp2);
38              break;
39          case abstr:
40              vars = varslivres(&(*r)->u.nodestruct.n2);
41              del(&vars, (*r)->u.nodestruct.n1->u.v);
42              break;
43      }
44
45      return vars;
46  }
```

A substituição é feita apenas em nós de abstração (como seria de esperar). Para facilitar, é feita a subsituição sempre, evitando complicar e ter de fazer mais que 1 passagem pela árvore para ver se todas as variáveis têm nomes diferentes. O algoritmo utilizado é algo como:

- Calcular variáveis livres dos termos dentro da abstração
- Pegar no nome da variável da abstração e encontrar um nome novo que ainda não esteja a ser utilizado nem por outras abstrações nem por variáveis livres
- Criar um nó var novo com esse novo nome
- Substituir na árvore pelo novo nó, aplicando o mesmo algoritmo nos nós de abstração encontrados mais abaixo pelos termos.

Novos exemplos

Os exemplos mostram várias aplicações e abstrações com o mesmo nome e duas variáveis livres com esse mesmo nome. É devolvido sempre termo alpha equivalente com todas as variáveis com nomes diferentes.

(λx.λy.λz.x y z) (λx.x) (λy.y) x y

```
@trabalho1 $ ./lambda
(!x.!y.!z.x y z) (!x.x) (!y.y) x y
<- (!x.!y.!z.x y z) (!x.x) (!y.y) x y
-> (!z.!b.!c.z b c) (!d.d) (!e.e) x y
```

(λx.(λx.x) x) x

```
@trabalho1 $ ./lambda
(!x.(!x.x) x) x
<- (!x.!x.x x) x
-> (!y.!z.z y) x
```

(λx.x) x (λx.(λx.x) (λx.x))

```
@trabalho1 $ ./lambda
(!x.x) x (!x.(!x.x) (!x.x))
<- (!x.x) x (!x.!x.x !x.x)
-> (!y.y) x (!z.!a.a !b.b)
```

(λa.(λa.(λa.(λa.a)a)a)a) a

```
@trabalho1 $ ./lambda
(!a.(!a.(!a.(!a.a)a)a)a) a
<- (!a.!a.!a.!a.a a a a) a
-> (!b.!c.!d.!e.e d c b) a
```

Bibliografia consultada

Slides das aulas teóricas, slides das aulas de compiladores.