Implementation of TLC (Tiny Lambda Calculus)

WANG Hanfei

October 27, 2019

Contents

2017 级弘毅班编译原理课程设计第 5 次编程作业 (the parser of TLC)

1 Introduction

Our goal is the effective implementation of the programming language TLC (Tiny Lambda Calculus) by using the closure.

Lambda calculus is a formal system in mathematical logic and computer science for expressing computation by way of variable binding and substitution (see `https://en.wikipedia.org/wiki/Lambda_calculus`).

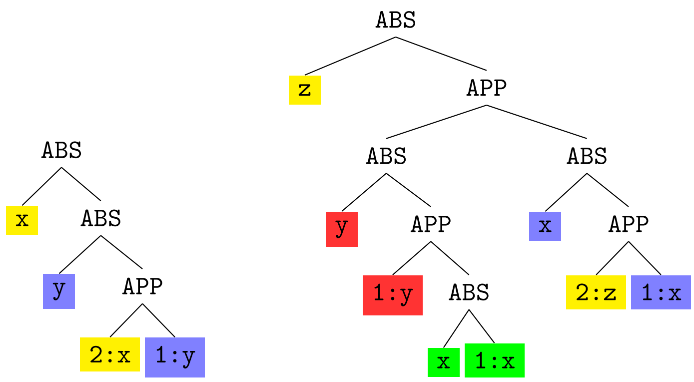
It is computation model of Functional Programming (see L. Paulson's lecture lambda.pdf, or my lecture lambda_lecture.pdf, and try lambda reducer at `http://www.itu.dk/people/sestoft/lamreduce/index.html`).

1.1 Specification of the language TLC

the syntax of TLC can be described as:

```
lines : lines decl
      | decl
      ;
decl : LET ID '=' expr ';'
     | expr ';'
     ;
expr : INT
     | ID
     | IF expr THEN expr ELSE expr FI
     | '(' expr ')'
     | '@' ID '.' expr
     | expr expr
     ;
```

where `@x.M` is the abstraction (instead of "λ" in lambda calculus for input). `M N` is the application. the conditional construct is specially added for the lazy evaluation of the conditional lambda terms (see `https://en.wikipedia.org/wiki/Lazy_evaluation`). `let X = M` is so called *let-binding* which binds the variable `X` with lambda term `M`. so any free occurrence of `X` in next context will referred to `M`. the application is left associative. and the precedence from low to high is: conditional construct, abstraction



(a) `@x.@y.x y` (b) `@z.(@y.y(@x.x))(@x.z x)`

Figure 1: AST with binding deepth (the first number of ID node)

and application.

see `lexer.l` and `grammar.y` in detail.

1.2 Abstract syntax trees

We use De Brujin index for the AST, it will replace the binding variable by the *binding depth*. Ex. `@x.@y.x` is `@x.@y.2`, `@z.(@y.y(@x.x))(@x.z x)` is `@z.(@y.1(@x.1))(@x.2 1)` (see Figure 1). It will be the key to access the closure environment in the implementation. the free occurence of variable is strictly forbidden in TLC.

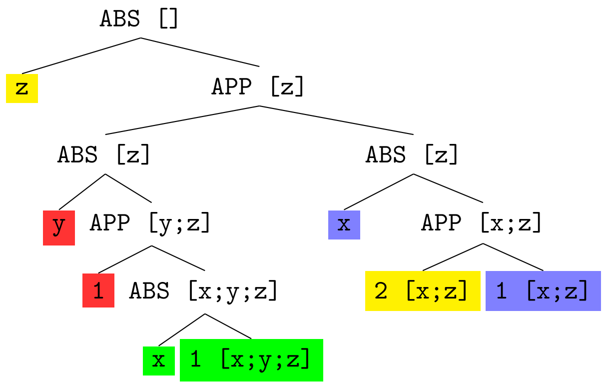
```
typedef enum {CONST=1, VAR=2, COND=3, ABS=4, APP=5} Node_kind;

typedef struct Ast {
  Node_kind kind;
  int value; /* for CONST and De Brujin index */
  struct Ast  *lchild, /*  for variable name and
                           abstraction variable
                         & apply function  body*/
    *rchild; /* for abstraction body  and app argument*/
  struct Ast *cond; /* for condition */
} AST;
```

1.3 Binding Deepth

to find the binding deepth, we use the static stack `char *name_env[MAX_ENV]` with the cursor `int current` (`tree.c`) to store the abstraction level. each time enter AST with `ABS` node, we push the abstraction name in the stack, increase `current` for the next, and popup by decreasing `current` after leave the abstraction body. each time a variable encountered in the abstraction body, `find_deepth()` will return the number of the depth in stack when first occurrence is found, see Figure 2.

```
int find_deepth(char *name)
{
  int i = current - 1;
  while (i + 1) {
    if (strcmp(name, name_env[i]) == 0) return current - i ;
    i--;
  }
  printf("id %s is unbound!\n", name);
  exit (1);
}
```



(a) `@z.(@y.y(@x.x))(@x.z x)`

Figure 2: Binding deepth

1.4 Primitive operations and let-bindings

in fact, let-bindings is a syntactic sugar of abstraction and application. so

```
let I = @x.x;
I 3;
```

can be interpreted as `(@I.(I 3))(@x.x)`.
So we will store all names of let-bindings in `name_env[]`.
the arithmetic operations can be treated as predefined let-bingding's name, `name_env[]` is prestored the following prefined functions and `current` initialized with `6` (see `grammar.y`):

```
char *name_env[MAX_ENV] = {"+", "-", "*", "/", "=", "<"};
int current = 6;
```

to the above binary operators work correctly in λ-calculus, its should interpret as `@x.@y.op x y`, that is prefix notations! so we will write `+ * 2 3 4` instead of `2 * 3 + 4`. hence

```
+ * 2 3 4
```

will be parsed as `((((+:6)(*:4))2)3)`.

when the following statement is parsed:

```
let I = @x.x;
```

`I` will be stored in `name_env[current]`. and we also store the AST of `@x.x` in the global `AST *ast_env[MAX_ENV]` (all defined in `grammar.y`) for the further uses. After `I` parsed, `name_env` and `ast_env` will be:

```
name_env[MAX_ENV] = {"+", "-", "*", "/", "=", "<", "I"}
ast_env[MAX_ENV] = {NULL, NULL, NULL, NULL, NULL, NULL, @x.(1:x)}
```

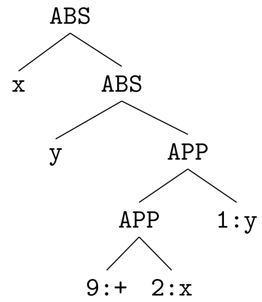
if we declare `PLUS` by input:

```
let PLUS = @x.@y. + x y;
```

the parser will generate the `(@x.(@y.(((+:9)(x:2))(y:1))))`. see Figure 3. In fact, after the parser enter the abstraction body `+ x y`, `name_env[]` will be:

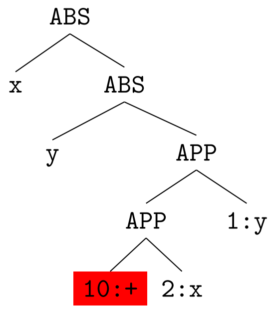
```
name_env[MAX_ENV] = {"+", "-", "*", "/", "=", "<", "I", "x", "y"}
```

so `find_deepth("+")` will return 9, `find_deepth("x") = 2`, and `find_deepth("y") = 1`.
after finish parsing, `name_env[]` changed to:



(a) `@x.@y.+ x y`

Figure 3: AST of `PLUS`



(a) `@x.@y.+ x y`

Figure 4: AST of `PLUS2`

```
name_env[MAX_ENV] = {"+", "-", "*", "/", "=", "<", "I", "PLUS"}
```

if we continue define `PLUS2` by input:

```
let PLUS2 = @x.@y + x 2;
```

the parser will generate the `(@x.(@y.(((+:10)(x:2))(y:1))))`. please remark that the binding deepth of `+` changed to 10 (see Figure 4). this is because the parsing of `PLUS2` is based with the new stack top `"PLUS"` of `name_env[]`, the the relative place of `"+"` is increased by `1`.
after `PLUS2`, `name_env[]` changed to:

```
name_env[MAX_ENV] = {"+", "-", "*", "/", "=", "<", "I", "PLUS", "PLUS2"}
```

1.5 two meanings of "="

the operators `"+"`, `"-"`, ..., `"="` will be recognized as normal `ID` with their binding deepth. but `"="` is also used as a single character token `'='` in the let-bingding, like `"let I = ..."`. To tell lexer return different tokens when `'='`is scanned, we use a global `int is_decl` (defined in `grammar.y`), and add a middle action in the `decl` production to active `is_decl`.

```
decl : LET {is_decl = 1; } ID '=' expr ';' {...}
```

and desactive it after return '=' in `lexer.l`:

```
"=" {
      char *id;
      if (is_decl) {is_decl = 0; return '='; }
      id = (char *) smalloc(yyleng + 1);
      strcpy(id, yytext);
      yylval = make_string(id);
      return ID;
}
```

1.6 output

We use the LaTeX graphic system tikz/pgf (`https://sourceforge.net/projects/pgf/`) and tikz-qtree (`https://ctan.org/pkg/tikz-qtree`) to illustrate AST. `printtree(AST *)` transforms the AST to LaTeX commands and store it in the file `expr.tex` which is the included file of `exptree.tex`. "`pdflatex exptree.tex`" generates the pdf of the AST (see `exptree.pdf`).

1.7 TODO

Completing `grammar.y` file to generate the AST for each lambda expression input, and output the AST to the file `expr.tex` by call `printtree(AST *)`. you can use lambda expression in `library.txt` to test your program.

please send grammar.y as attached file to `mailto:hfwang@whu.edu.cn?subject=ID(05)` where the `ID` is your student id number.

-hfwang October 27, 2019