Implementation of TLC (Tiny Lambda Calculus)

WANG Hanfei

December 18, 2017

Contents

1 Introduction **2**
- 1.1 Specification of the language LAMBDA . . . 2
- 1.2 Abstract syntax trees . . . 2
- 1.3 Binding Deepth . . . 2
- 1.4 Primitive operations . . . 3
- 1.5 output . . . 5

2 Typing **5**
- 2.1 Syntax-directed type synthesiser . . . 5
- 2.2 step by step of type synthesis . . . 6
  - 2.2.1 Example 1: `@x.@y.x y` (`MN`: `M` is a type variable) . . . 8
  - 2.2.2 Example 2: `@x.(@x.@y.x y) x` (`M` is arrow and `N` is type variable) . . . 10
  - 2.2.3 Example 3: `(@x.@y.x y)(@x.x)` (`M` and `N` are both arrow) . . . 11
  - 2.2.4 Example 4: `@x.x x` . . . 12
  - 2.2.5 Example 5: `let MY = @x.@y.x y;` . . . 12
  - 2.2.6 Example 6: `@x.MY x` . . . 13
  - 2.2.7 Example 7: recursions . . . 15
- 2.3 Church encoding and Type . . . 15

3 TODO **16**
- 3.1 Unification algorithm . . . 16
- 3.2 the semantic rules of type synthesis . . . 17
- 3.3 Typing . . . 17
- 3.4 Memory leaks . . . 18

4 Evaluation **18**
- 4.1 Evaluation of Call-by-value . . . 19
  - 4.1.1 Example 1: `(@x.@y.x y)(@x.x)` . . . 19
  - 4.1.2 Example 2: `(@x.@y.x y)(@x.x)3` . . . 20
  - 4.1.3 Example 3: `+ 3 4` . . . 20
  - 4.1.4 Example 4: `let MY = @x.+ x 3` . . . 20
  - 4.1.5 Algorithm . . . 22
- 4.2 Evaluation of Call-by-name . . . 23
  - 4.2.1 Example 1: `(@x.@y.x y)(@x.x) 3` . . . 24
  - 4.2.2 Example 2: `+ 2 (* 3 4)` . . . 25
  - 4.2.3 Example 3: `let MY = @x.+ x 3` . . . 25
  - 4.2.4 Example 4: fixed-point combinators . . . 26
  - 4.2.5 Lazy infinit list . . . 28
- 4.3 Memory Leaks . . . 28

5 TODO **29**

2015 级弘毅班编译原理课程设计第 6 次编程作业 (CBV and CBN 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).

1.1 Specification of the language LAMBDA

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. and the conditional construct is specially added for the lazy evaluation of the conditional lambda terms. the application is left associative. and the precedence from low to high is: conditional construct, abstraction and application.
see `lexer.l` and `grammar.y` in detail.

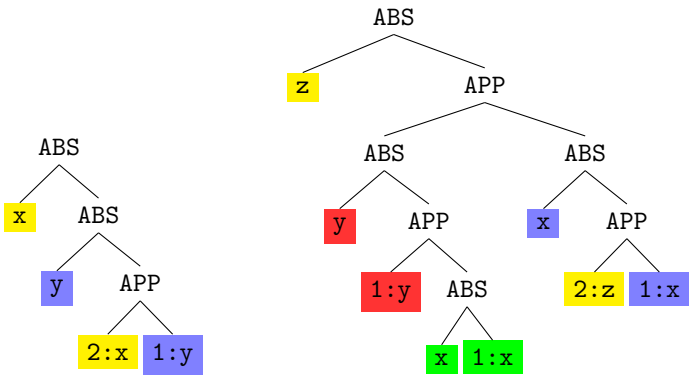
1.2 Abstract syntax trees

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;
```

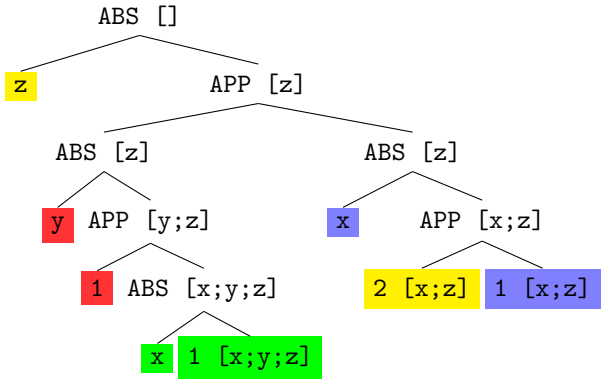
1.3 Binding Deepth

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 deepth in stack when first occurrence is found, see Figure 2.



(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)



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

Figure 2: Binding deepth

```
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);
}
```

1.4 Primitive operations

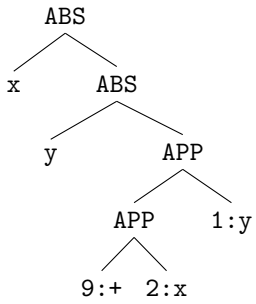
`char *name_env[]` will also store the name of the declaration. so when the following statement is parsed:

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

`I` will 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 (typing).
to support the arithmetic operations, `name_env[]` is prestored the following prefined functions:

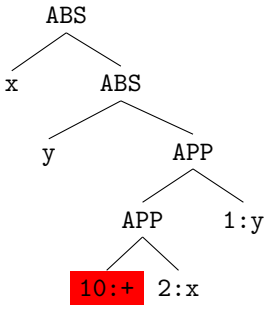
```
char *name_env[MAX_ENV] = {"+", "-", "*", "/", "=", "<"};
```

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`.



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

Figure 3: AST of `PLUS`



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

Figure 4: AST of `PLUS2`

the binding deepth is also the key to access the function defined in the declaration. so when `I` is declared, the `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)}
/* AST of operators is not needed for typing */
```

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:

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

if we continue define `PLUS2` by input:

```
let PLUS = @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"}
```

the operators "+", "-", ... must be scanned as normal ID with their binding deepth. but "=" is also used as a single character token in the declaration like "let I = ...". we use a global int is_decl (defined in grammar.y) to tell the lexer if "=" should return '=' or ID, and add a middle action in the decl production to active is_decl:

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

deactive each time before 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;
}
```

1.5 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).

2 Typing

> Well-typed programs cannot go wrong — Robin Milner

(see https://en.wikipedia.org/wiki/Type_safety)

2.1 Syntax-directed type synthesiser

As we know, if we admit the "x x" in lambda term, this will cause the Russell's paradox (see L. Paulson's lecture " Foundation of Functional Programming" (PP. 23). to avoid this paradox, we can annotate each lambda term with a type, and if such type can't be established, the term will be rejected.

as example, if "x x" the first "x" should be a function type of form "A -> B" (A and B are any sets, we call them *type variables*) if it can be applied by an argument, the second "x" must be the type of the domain of first x, that is A. (see https://en.wikipedia.org/wiki/Simply_typed_lambda_calculus or Pierce's Book "Types and Programming Languages", Ch. 9: Simply Typed Lambda-Calculus, in our compiler_cd directory). So the type constraint is the equation of type:

```
A =  A -> B
```

where A is *type variable* in the type set defined recursively as:

1. type constant int is a type.
2. X, Y, Z, ..., the alphabets of *type variable* are the type.
3. if A and B are any type, then A -> B is a type. (so X -> X, X -> Y, (X -> Y) -> Z, ..., are types).

because type variable A appears in right side of the above equation, we have not solution for it.

but the lambda term (@x.x)(@x.x) can be type as X -> X. the first (@x.x) (denoted by alpha) can be type as (A -> A) and the second (@x.x) (denoted by beta) can be typed as (B -> B). for the term "alpha beta" have sense, the term alpha must have type function with domain B -> B, so the type equation is:

```
A =  B -> B
```

so `A = C -> C` and `B = C` is the solution.

and the *domain* type (left side of the arrow) of `alpha` is `C -> C`, and the type of `(@x.x)(@x.x)` is the type of the *range* (right side of the arrow) of `alpha`, so `C -> C`. Remark the type variable may be changed to any other type.

the above equation has infinite solution like:

```
A = (D -> D) -> (D -> D) and B = D -> D
A = ((D -> D) -> D) -> ((D -> D) -> D) and B = ((D -> D) -> D)
......
```

but all the above solution can be obtained by the substitution of `C` in the first solution. the first solution is so called *most general*.

the difference of the above 2 term `(x x)` and `(@x.x)(@x.x)` is the 2 occurrence of `x` in the first one are the same `x`. but the second are not the same.

Our goal is establish the most general type of any given lambda term if it has, or announce the type error if not. the method is *syntax-directed type synthesis*.

2.2 step by step of type synthesis

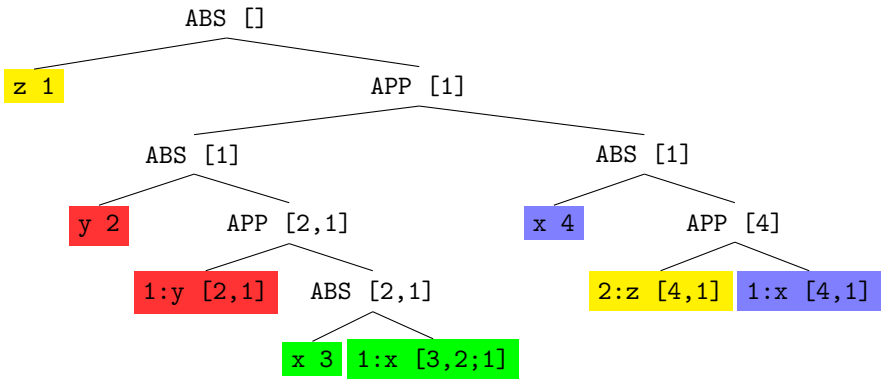
the type structure is defined in `type.h` as:

```
typedef enum { Typevar = 1,  Arrow = 2, Int = 3 } Type_kind;
  /* for type tree node */

typedef struct type {
  int index;  /* for coding type variable */
  Type_kind kind;
  struct type * left, *right;
} Type;

typedef Type * Type_ptr;

typedef struct type_env{
  int redirect;   /* for unification use */
  Type_ptr type;   /* pointer to the type tree structure */
} Type_env;

typedef Type_env * Type_env_ptr;

extern Type_ptr global_type_env[MAX_ENV];
  /* like name_env[] and ast_env[], global_type_env[]
     will store the type for the declared lambda term */
```

1. each type variable is coded with a unique index as its type, and the each node of type tree is also coding with this index.

2. store each type tree node in the typing environment `Type_env type_env[MAXNODE]` (see `type.c`), and any type of the index `i` will store in `type_env[i]`.

3. with the De Brujin index 1.2, a lambda variable in the abstraction is represented by the binding deepth. to access the corresponding type variable, we can use a stack, like build the AST. each time enter an abstraction body, the corresponding abstraction variable is push to stack, the type of the variable encountered in the body, is just the `n`-th element from the top of the stack, where `n` is the binding deepth. in Figure 5, the preorder tree traversal of AST generate a new type variable (indexed from `1` to `4` for each abstraction, and push it in the stack when enter the abstraction body. for the `x` in the subterm `@x.x`, the binding deepth is `1`, so the first element (that's `3`) from the



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

Figure 5: Binding deepth for retrieve the type

top stack `[3,2,1]` (stack top is on the **left**) is the corresponding type. for the `z` in the subterm `@x.z x`, the binding deepth is `2`, so the second element (that's `1`) from the top stack `[4,1]` is the corresponding type.

the stack is implemented by a dynamic list:

```
typedef struct varlist {
char *var_name;
struct varlist * next;
} Var_list;
typedef Var_list * Var_list_ptr;
```

4. normally, we should initialze the stack with all elements in `global_type_env[]` of predefined lambda term. it's very heady do to it, if the predefined terms are too many. we split the stack into two parts, the all abstraction of the current term in the dynamic list `Var_list_ptr abs`, and all predefined term in `global_type_env[]` with the top `current - 1` (see 1.3).

in the next, an entry `type_env[n]` is denoted as `n(redirect, type)`, `type` is either `n` (type variable), either `int` (type constant), or `(x -> y)` (arrow), like

`1(1, (3 -> 4))`

Example of `type_env[]`

`type_env[] = [0(0,int), 1(1,(3 -> 4)), 2(2,2), 3(2,3), 4(4,4)]`

where `3(2,3)` means type variable `3` is *redirect* to `2`.

we deonote `M |== T` if the lambda term `M` has the most general type `T`.

the global environments are setting as:

```
name_env[] = {"+", "-", "*", "/", "=", "<"}
global_type_env[] = [0(int->(int->int)), 1(int->(int->int)), 2(int->(int->int)),
                     3(int->(int->int)), 4(int->(int->int)), 5(int->(int->int))]
current = 6  /* stack top + 1 */
```

the type constant `int` always has the index `0`.

Each time typing a lambda term, `type_env[]` will be set to

```
type_env[] = [0("",0,0,int)]
nindex = 1 /* next index for type variable */
```

its can be done by

```
void init_type_env()
{
  int i = 0;

  type_env[0] = &inttype_entry;
  type_env[0] -> type = & inttype;

  while (i < INIT_POS) {
    new_env();
    global_type_env[i] =
      storetype(make_arrowtype( &inttype, make_arrowtype(&inttype, &inttype)));
    /*  int -> (int -> int) */
    i++;
  }
  return;
}
```

We will use a serie of the examples to stepwise the typing processus by the **recursive tree traversal** of AST.

2.2.1 Example 1: `@x.@y.x y` (`MN: M` is a type variable)

Like computing the binding deepth, the **preorder tree traversal** of AST find the abstraction stack for each AST node. the following **postorder traversal** type the lambda term. here is the postorder

step 1

```
top = 8        /* the stack top index = current + lenght(abs) */
abs: [2,1]     /* the stack of abstraction */
type_env[] = [0(0,int), 1(1,1), 2(2,2)]
(x:2) |== 1    /* obtain by get 2-th from top of abs */
```

setp 2

```
top = 8:
abs: [2,1]
type_env[] = [0(0,int), 1(1,1), 2(2,2)]
(y:1) |== 2   /* obtain by get 1-th from top of abs */
```

step 3

```
top = 8:
abs: [2,(3 -> 4)]
type_env[] = [0(0,int), 1(1,(3 -> 4)), 2(2,2), 3(2,3), 4(4,4)]
((x:2)(y:1)) |== 4
```

if the term `x y` has sense, the `x` must have the arrow type, but isn't the case. fortunately, `x` is a type variable, it can be change to any type. function `get_instance(Type_ptr)` will do that:

```
Type_ptr get_instance(Type_ptr type_tree)
{
  Type_ptr p = final_type(type_tree);
     /* p must be a no redirect type, see below */
  p -> kind = Arrow;
  p -> left = make_vartype(0, 0);
  p -> right = make_vartype(0, 0);
  return p;
}
```

it rewrites the type of the index `1` to an arrow type. and generate 2 type variables of `index` `3` and `4`. after `get_instance(1)`, `x` changed to type `(3 -> 4)`. and `y` is of type `2`. to the application `x y` have the sense, the domain type `2` of `x` and type `3` of `y` **must be** the same. We say they should *unified*. we can do it by changing `3` to `2`. the problem is that all typing envrionment which refers `3` must be changed to `2`. that is hard job (search all `type_env[]`, replacing the index of `3` with `2`, just like update the primary key in the database, you must alter all foreign keys that refer the updated primary key)! To simply this tedious job, we just redirect the entry of `3` in `type_env[]` from itself (`3`) to `2`. so we have `3(2,3)` in `type_env[]` where the first component change from `3` to `2` (points to `2`). this work can be done by the **side-effect** function `unify_leaf()` (it change the global `type_env[]`!).

```
void unify_leaf(Type_ptr t1, Type_ptr t2)
{
  int index1 =  (t1 -> index);
  int index2 =  (t2 -> index);

  if (index1 != index2) {
    type_env[index1] -> redirect = index2;
  }
  return;
}
```

a type variable `n(m, t)` in `type_env[]` is called *final* iff `n == m`, ifnot we call it *non-final*. `int final_type()` will across the redirect chain to get the final type. be careful, each time we access the type node which is not-final, retrieve the final type.

```
int final_index (int index)
{
  int i = index;
  if (type_env[i] == NULL) return -1;
  if ( type_env[i] -> type -> kind == Arrow )
    return i;
  if (i == (type_env[i] -> redirect))
    return i;
  return final_index(type_env[i] -> redirect );
}

/* return final type node for a giving Typevar node */
Type_ptr final_type(Type_ptr t)
{
  int i;
  i = final_index(t -> index);
  if (i == -1) return NULL;
  return type_env[i] -> type;
}
```

step 4

```
top = 7:
abs: [(3 -> 4)]
type_env[] = [0(0,int), 1(1,(3 -> 4)), 2(2,2), 3(2,3), 4(4,4), 5(5,(2 -> 4))]
(@y.((x:2)(y:1))) |== (2 -> 4)
```

`y` is of type `2`, and `x y` is of type `4`, the abstraction of `@y.x y` will be typed as an arrow type with a new generated index `5`.

step 5

```
top = 6:
abs: []
type_env[] = [0(0,int), 1(1,(3 -> 4)), 2(2,2), 3(2,3), 4(4,4), 5(5,(2 -> 4)),
              6(6,((3 -> 4) -> (2 -> 4)))]
(@x.(@y.((x:2)(y:1)))) |== ((3 -> 4) -> (2 -> 4))
```

`printtype(Type_ptr)` will recode the final type variable to `A - Z`, so `2` as `A`, `4` as `B`, then output

```
(@x.(@y.((x:2)(y:1)))) |==  ((A -> B) -> (A -> B))
```

step 6

`printtype(Type_ptr)` will recode the final type variable to `A - Z`, so `2` as `A`, `4` as `B`, then output

```
(@x.(@y.((x:2)(y:1)))) |==  ((A -> B) -> (A -> B))
```

step 7

after the typing processus finished, all type that we dynamically allocated must be freed. because every type we maked has an index in `type_env[]`, so we can free all of them easily without any memory leak by

```
void new_env(void)
{
  int i;
  for (i = 1; i < nindex; i++) {
    sfree(type_env[i] -> type);
    sfree(type_env[i]);
    /* redefine free as sfree (in emalloc.c) for
       gprofile the call frequences of free() */
  }
  for (i = 0; i < order; i++)
    index_order [i] = 0;

  nindex = 1;
  order = 1;
  step = 0;
}
```

2.2.2 Example 2: `@x.(@x.@y.x y) x` (`M` is arrow and `N` is type variable)

the first 5 steps is the same as Example 1 with all indexes increased by `1`.
step 5

```
top = 7:
abs: [1]
type_env[] = [0(0,int), 1(1,1), 2(2,(4 -> 5)), 3(3,3), 4(3,4), 5(5,5), 6(6,(3 -> 5)),
              7(7,((4 -> 5) -> (3 -> 5)))]
(@x.(@y.((x:2)(y:1)))) |== ((4 -> 5) -> (3 -> 5))
```

step 6

```
top = 7:
abs: [1]
type_env[] = [0(0,int), 1(1,1), 2(2,(4 -> 5)), 3(3,3), 4(3,4), 5(5,5), 6(6,(3 -> 5)),
              7(7,((4 -> 5) -> (3 -> 5)))]
(x:1) |== 1
```

step 7

```
top = 7:
abs: [1]
type_env[] = [0(0,int), 1(2,1), 2(2,(3 -> 5)), 3(3,3), 4(3,4), 5(5,5), 6(6,(3 -> 5)),
              7(7,((3 -> 5) -> (3 -> 5)))]
((@x.(@y.((x:2)(y:1))))(x:1)) |== (3 -> 5)
```

in this time, the function ((@x.(@y.((x:2)(y:1)))) of the application is already an arrow type ((4 -> 5) -> (3 -> 5)) where (4 -> 5) is of the index 2, the argument is type variable 1, we can just redirect 1 to unify the domain type of the arrow and the argument type. so we have the side-effect 1(2,1) in type_env[]

```
void unify_leaf_arrow(Type_ptr leaf, Type_ptr t)
{
  int index = leaf -> index;
  type_env[index] -> redirect = t -> index;
  return;
}
```

step 8

```
top = 6:
abs: []
type_env[] = [0(0,int), 1(2,1), 2(2,(3 -> 5)), 3(3,3), 4(3,4), 5(5,5), 6(6,(3 -> 5)),
              7(7,((3 -> 5) -> (3 -> 5))), 8(8,(1 -> (3 -> 5)))]
(@x.((@x.(@y.((x:2)(y:1))))(x:1))) |== (1 -> (3 -> 5))
```

remark that the domain type 1 of 8(8,(1 -> (3 -> 5))) is non-final, the redirect final type is 2(2,(3 -> 5)). so

step 9

```
(@x.((@x.(@y.((x:2)(y:1))))(x:1))) |== ((A -> B) -> (A -> B))
```

2.2.3 Example 3: (@x.@y.x y)(@x.x) (M and N are both arrow)

the first 5 steps is the same as Example 1.

step 5

```
top = 6:
abs: []
type_env[] = [0(0,int), 1(1,(3 -> 4)), 2(2,2), 3(2,3), 4(4,4), 5(5,(2 -> 4)),
              6(6,((3 -> 4) -> (2 -> 4)))]
(@x.(@y.((x:2)(y:1)))) |== ((3 -> 4) -> (2 -> 4))
```

step 6

```
top = 7:
abs: [7]
type_env[] = [0(0,int), 1(1,(3 -> 4)), 2(2,2), 3(2,3), 4(4,4), 5(5,(2 -> 4)),
              6(6,((3 -> 4) -> (2 -> 4))), 7(7,7)]
(x:1) |== 7
```

step 7

```
top = 6:
abs: []
type_env[] = [0(0,int), 1(1,(3 -> 4)), 2(2,2), 3(2,3), 4(4,4), 5(5,(2 -> 4)),
              6(6,((3 -> 4) -> (2 -> 4))), 7(7,7), 8(8,(7 -> 7))]
(@x.(x:1)) |== (7 -> 7)
```

step 8

```
top = 6:
abs: []
type_env[] = [0(0,int), 1(1,(2 -> 4)), 2(7,2), 3(2,3), 4(7,4), 5(5,(2 -> 4)),
              6(6,((2 -> 4) -> (2 -> 4))), 7(7,7), 8(8,(7 -> 7))]
((@x.(@y.((x:2)(y:1))))(@x.(x:1))) |== (2 -> 4)
```

in this time, the argument of function is also function `(7 -> 7)` which should be unified with `(2 -> 4)`. the unification can be done by unify both domain type and range type of the arrow. `unify(2, 7)` is the case `unify_leaf()` which redirect `2` to `7`. `unify(4, 7)` redirect `4` to `7` also.

step 9

```
((@x.(@y.((x:2)(y:1))))(@x.(x:1))) |== (A -> A)
```

2.2.4 Example 4: `@x.x x`

step 1

```
top = 7:
abs: [1]
type_env[] = [0(0,int), 1(1,1)]
(x:1) |== 1
```

step 2

```
top = 7:
abs: [1]
type_env[] = [0(0,int), 1(1,1)]
(x:1) |== 1
```

step 3

```
top = 7:
abs: [(2 -> 3)]
type_env[] = [0(0,int), 1(1,(2 -> 3)), 2(2,2), 3(3,3)]
((x:1)(x:1)) |== NULL
type A and type (A -> B) can't be unified!
```

the function type is a type variable `1`. `get_instance(1)` rewrite to an arrow. the side-effect change the argument type (second `x`) to the same arrow. `unify(2, 1)` is the case `unify_leaf_arrow(2, 1)`, but if we redirect `2` to `1`. but `2` is in type `1(1, (2 -> 3))`. it's not typable this will also cause the cyclic chain of redirection. it is strictly forbidden. `is_occur_node(2, (2 -> 3))` checks if such case. if the occurrence take place, report typing error and return `NULL`.

```
int is_occur_node(int index, Type_ptr type_tree)
{
  int i = index;
  if (type_tree == NULL) return 1;

  switch (type_tree -> kind) {
  case Typevar:
    return  type_env[type_tree -> index] -> redirect == i;
  case Arrow:
    /* left and right may be not final!!! */
    return is_occur_node (i, final_type(type_tree -> left)) ||
      is_occur_node(i, final_type(type_tree -> right));
  case Int:
    return 0;
  }
}
```

2.2.5 Example 5: `let MY = @x.@y.x y;`

the first 5 steps is the same as Example 1.

step 5

```
top = 6:
abs: []
type_env[] = [0(0,int), 1(1,(3 -> 4)), 2(2,2), 3(2,3), 4(4,4), 5(5,(2 -> 4)),
              6(6,((3 -> 4) -> (2 -> 4)))]
(@x.(@y.((x:2)(y:1)))) |== ((3 -> 4) -> (2 -> 4))
```

step 6

We will store the name `MY` in `name_env[6]` and the AST `(@x.(@y.((x:2)(y:1))))` in `ast_env[6]`, and the type in `global_type_env[6]`. if the next lambda term referes the `MY`, we should restore the type of `MY` in new `type_env[]`. `storetype(Type_ptr t)` will reindex the arrow type with `0` and the final type variable from `1` to `n` if the type tree has `n` different leaves.

```
/* generate type_env independant  type tree */
Type_ptr storetype(Type_ptr tree)
{
  if (tree == NULL) return;
  switch ( tree -> kind ) {
  case Int:  return &inttype;
  case Typevar: {
    int i = final_index(tree -> index);
    Type_ptr t = type_env[i] -> type;
    switch (t -> kind) {
    case Int:   return &inttype;
    case Arrow:
      tree -> left = t -> left;
      tree -> right = t -> right;
      break;
    default: {
      int offset = find_index(i); /* reindex the type variable */
      Type_ptr tmp;
      if (offset == 0) {
        return &inttype;
      }
      tmp = (Type_ptr) smalloc(sizeof(Type));
      tmp -> index = offset;
      tmp -> kind = Typevar;
      tmp -> left = tmp -> right = NULL;
      return tmp;
    }
    }
  }
  }
  {
    Type_ptr tmp = (Type_ptr) smalloc(sizeof(Type));
    tmp -> index = 0;
    tmp -> kind = Arrow;
    tmp -> left = storetype(tree -> left);
    tmp -> right = storetype(tree -> right);
    return tmp;
    }
}
```

so `global_type_env[6] = (1 -> 2) -> (1 -> 2)`, and the cursor `currrent` for the next abstraction or definition is increased to 7.

2.2.6 Example 6: `@x.MY x`

it is the same of Example 2, but the subterm `(@x.@y.x y)` is predefined. we should access the correct place in `global_type_env[6]` and restore the type of `MY` in the new `type_env[]`.

step 1

```
top = 8:
abs: [1]
type_env[] = [0(0,int), 1(1,1), 2(2,2), 3(3,3), 4(4,(3 -> 2)), 5(5,(3 -> 2)),
              6(6,((3 -> 2) -> (3 -> 2)))]
(MY:2) |== ((3 -> 2) -> (3 -> 2))
```

`MY` is of binding deepth `2` which is greater than the stack length `abs`, so it is a predefined name. because the `glolal_env[current - 1]` is the extension of the stack, we can get it by `top - 2`. `get_n_th([1], 2, 8)` will restore the type of `MY` as `6(6,((3 -> 2) -> (3 -> 2)))` with new series of indexes from `2` to `6`.

```
Type_ptr get_n_th_from_global(int i)
{
  /* if is the fixed-point combinator, we will
     assign it with the type (A -> A) -> A.
     Z, Y and rec is defined in library.txt */
  if (strcmp(name_env[i], "Z") == 0 ||
      strcmp(name_env[i], "Y") == 0 ||
      strcmp(name_env[i], "rec") == 0) {
    return make_rec_type();
  }
  return restoretype(global_type_env[i]);
  /* restoretype will reindex the type variable in new type_env[] */
}

/* pos is the current top of name_env[] */
/* n is the binding deepth of the lambda variable */
Type_ptr get_n_th(Var_list_ptr list, int n, int pos)
{
  int i = 0;
  while (i != n - 1 && list != NULL) {
    list = list -> next;
    i++;
  }

  /* is an abstraction */
  if (i == n - 1  && list != NULL)
    return list -> type_var;

  /* is a predefined name */
  if ((pos - n) >= 0)
    return get_n_th_from_global(pos - n);

  printf("wrong access global type env\n");
  exit (1);
}
```

the following steps are as Example 2.

step 2

```
top = 8:
abs: [1]
type_env[] = [0(0,int), 1(1,1), 2(2,2), 3(3,3), 4(4,(3 -> 2)), 5(5,(3 -> 2)),
              6(6,((3 -> 2) -> (3 -> 2)))]
(x:1) |== 1
```

step 3

```
top = 8:
abs: [1]
type_env[] = [0(0,int), 1(5,1), 2(2,2), 3(3,3), 4(4,(3 -> 2)), 5(5,(3 -> 2)),
              6(6,((3 -> 2) -> (3 -> 2)))]
((MY:2)(x:1)) |== (3 -> 2)
```

step 4

```
top = 7:
abs: []
type_env[] = [0(0,int), 1(5,1), 2(2,2), 3(3,3), 4(4,(3 -> 2)), 5(5,(3 -> 2)),
              6(6,((3 -> 2) -> (3 -> 2))), 7(7,(1 -> (3 -> 2)))]
(@x.((MY:2)(x:1))) |== (1 -> (3 -> 2))
```

step 5

```
(@x.((MY:2)(x:1))) |== ((A -> B) -> (A -> B))
```

2.2.7 Example 7: recursions

the fixed-point combinator (https://en.wikipedia.org/wiki/Fixed-point_combinator) can be predefined as

```
let Y=@f.(@x.f(x x))(@x.f(x x));
let Z=@f.(@x.f(@y.(x x)y))(@x.f(@y.(x x)y));
```

because x apply itself, it can't be typable, if we input:

```
@f.(@x.f(x x))(@x.f(x x));
typing step 1 and top = 9:
abs: [2,1]
type_env[] = [0(0,int), 1(1,1), 2(2,2)]
(f:2) |== 1
typing step 2 and top = 9:
abs: [2,1]
type_env[] = [0(0,int), 1(1,1), 2(2,2)]
(x:1) |== 2
typing step 3 and top = 9:
abs: [2,1]
type_env[] = [0(0,int), 1(1,1), 2(2,2)]
(x:1) |== 2
type A and type (A -> B) can't be unified!
typing step 4 and top = 9:
abs: [(3 -> 4),1]
type_env[] = [0(0,int), 1(1,1), 2(2,(3 -> 4)), 3(3,3), 4(4,4)]
((x:1)(x:1)) |== NULL
```

to type the recursive function defined by the fixed-point combinator, we should give its the type `(A -> A) -> A`. so we have:

```
let fact = (Z (@f.@n. (if (= n 0) then 1 else (* n (f (- n 1))) fi)));
```

it will return

```
((Z:1)(@f.(@n.if(((=:7)(n:1))0)then1else(((*:9)(n:1))((f:2)(((-:10)(n:1))1))))))
|= (int -> int)
```

this can be done in `get_n_th_from_global(int i)` by checking if `name` of a global `i` is `Y`, `Z`, or `rec`. if so, just return the above type.

2.3 Church encoding and Type

In mathematics, Church encoding is a means of representing data and operators in the lambda calculus. The data and operators form a mathematical structure which is embedded in the lambda calculus. The Church numerals are a representation of the natural numbers using lambda notation. The method is named for Alonzo Church, who first encoded data in the lambda calculus this way (https://en.wikipedia.org/wiki/Church_encoding). You can see this encoding in library.txt.

1. Church numerals:

```
ZERO |== (A -> (B -> B))
ONE |== ((A -> B) -> (A -> B))
TWO |== ((A -> A) -> (A -> A))
......
FIVE |== ((A -> A) -> (A -> A))
```

2. Arithmetic operation:

```
ADD |== (((((A -> B) -> (C -> A)) -> ((A -> B) -> (C -> B)))
        -> (D -> E)) -> (D -> E))
SUB |== (A -> (((((((B -> (C -> B)) -> D) -> ((E -> (D -> F)) -> F)) ->
        (((G -> (G -> H)) -> H) -> ((I -> (J -> J)) -> K))) -> ((D -> E) ->
        (G -> K))) -> (A -> L)) -> L))
MULT |== (((A -> B) -> ((C -> (D -> D)) -> E)) -> (((((F -> G) -> (H -> F))
         -> ((F -> G) -> (H -> G))) -> (A -> B)) -> E))
PRED |== (((((A -> (B -> A)) -> C) -> ((D -> (C -> E)) -> E)) -> (((F -> (F -> G)) -> G)
         -> ((H -> (I -> I)) -> J))) -> ((C -> D) -> (F -> J)))
```

3. Booleans

```
(TRUE:65) |== (A -> (B -> A))
(FALSE:64) |== (A -> (B -> B))
IF |== ((A -> (B -> C)) -> (A -> (B -> C)))
OR |== (((A -> (B -> A)) -> (C -> D)) -> (C -> D)
AND |== ((A -> ((B -> (C -> C)) -> D)) -> (A -> D))
NOT |== (((A -> (B -> B)) -> ((C -> (D -> C)) -> E)) -> E)
GE |== (((((((A -> (B -> A)) -> C) -> ((D -> (C -> E)) -> E)) -> (((F -> (F -> G)) -> G)
       -> ((H -> (I -> I)) -> J))) -> ((C -> D) -> (F -> J))) ->
       (K -> ((L -> (M -> (N -> N))) -> ((O -> (P -> O)) -> Q)))) -> (K -> Q))
LE |== (A -> (((((((B -> (C -> B)) -> D) -> ((E -> (D -> F)) -> F)) ->
       (((G -> (G -> H)) -> H) -> ((I -> (J -> J)) -> K))) -> ((D -> E) -> (G -> K))) ->
       (A -> ((L -> (M -> (N -> N))) -> ((O -> (P -> O)) -> Q)))) -> Q))
EQ |== NULL
let LEQ = @m.@n.ISZERO (SUB m n);
LEQ |== (A -> (((((((B -> (C -> B)) -> D) -> ((E -> (D -> F)) -> F)) ->
        (((G -> (G -> H)) -> H) -> ((I -> (J -> J)) -> K))) -> ((D -> E) -> (G -> K)))
        -> (A -> ((L -> (M -> (N -> N))) -> ((O -> (P -> O)) -> Q)))) -> Q))
let EQ1 = @m.@n. AND (LEQ m n) (LEQ n m);
EQ1 |== NULL
```

EQ and EQ1 can't be typed!

4. Recursions

```
Y |== NULL
Z |== NULL
FACT |== NULL
SUM |== NULL
DIV |= NULL
fact |== (int -> int)
```

```
ACK |== ((((((A -> B) -> (A -> B)) -> C) -> (((((A -> B) -> (A -> B)) -> C) ->
        (C -> D)) -> D)) -> ((((E -> F) -> (F -> G)) ->
        ((E -> F) -> (E -> G))) -> H)) -> H)
```

`FACT` and `SUM` coding with Church numerals can't be typed, even added the axiom `Y |== (A -> A) -> A`. `fact` with lazy `if` is typable under the axiom. Ackermann function (https://en.wikipedia.org/wiki/Ackermann_function) is typable.

3 TODO

3.1 Unification algorithm

refer the unification algorithm of Dragon book (PP. 397) finish the side-effect unification algorithm

```
/* return 1 if unified; return 0 ifnot */
int unify(Type_ptr t1, Type_ptr t2)
{
  t1 =  simply(t1);
  t2 =  simply(t2);
  if (t1 == NULL || t2 == NULL) {
    printf("null type occur! typing error!\n");
    return 0;
  }
  switch (t1 -> kind) {
  case Int: {
     /* todo */
  }
  case Typevar: {
    /* todo */
  }
  case Arrow: {
    /* todo */
  }
  }
  return 1;
}
```

3.2 the semantic rules of type synthesis

each `AST T` has 3 attributes:

1. `T.top` : `= current + abstraction deepth`
2. `T.abs`: the stack of the type of the abstraction.
3. `T.type`: type

the typing will work with the global `type_env[]`, `name_env`, `global_type_env[]`, `nindex`, `current`...

AST	semantic rules
ROOT	ROOT.abs = [ ] /* empty stack */ ROOT.top = current
T = CONST n	T.type = int
T = VAR (n:x)	T.type = get_nth(T.abs, n, T.top)
T = ABS (x, T1)	x.type = make_vartype() T1.abs = add_list(x.type, T.abs) T.type = make_arrow(x.type, T1.type) T1.top = T.top + 1
T = COND(T1, T2, T3)	T1.abs = T.abs T1.top = T.top T2.abs = T.abs T2.top = T.top T3.abs = T.abs T3.top = T.top if (T1.type == int && unify(T2.type, T3.type)) T.type = T2.type else T.type = NULL
T = APP (T1, T2)	/* todo */ /* you should included this SDD in the file type.c */

3.3 Typing

the attributes T.abs and T.top are L-attributed. T.type is S-attributed. so we can solve them with recursive tree traversal. Implement the following typing function.

```
Type_ptr typing (Var_list_ptr abs, AST *t, int top)
{
  Type_ptr tmp; /* for store the return type */

  if (t == NULL) return NULL;

  switch (t -> kind) {
  case CONST: return make_inttype();
  case VAR: {
    tmp = get_n_th(abs, t -> value, top);
    break;
  }

  case ABS:  {
    /* todo */
  }

  case COND: {
    /* todo */
  }
  case APP: {
   /* todo */
  }
  }
  if (yyin == stdin) {
    printf("typing step %d and top = %d:\n", ++ step, top);
    print_abs(abs);
    print_env();
    print_expression(t, stdout);
    printf(" |== "); print_type_debug(tmp); printf("\n");
  }
  free_list(abs);
  return tmp;
}
```

Be careful: every recursive call of typing() should pass the copy of abs list by call Var_list_ptr list_copy (Var_list_pt).

3.4 Memory leaks

You program should have no memory leaks! you can test it by input multiple lines:

```
@m.m(@f.@n.n f(f(@f.@x.f x)))(@n.@f.@x.n f (f x));
```

then `gprof ./lambda`. the difference of `smalloc` and `sfree` should be the same like:

ONE INPUT of the above lambda term:

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call  name
 0.00      0.00     0.00      602     0.00     0.00  print_type_debug
 0.00      0.00     0.00      268     0.00     0.00  smalloc
 0.00      0.00     0.00      226     0.00     0.00  sfree
 ......
```

TWO INPUT of the above lambda term:

```
 %   cumulative   self              self     total
time   seconds   seconds    calls  Ts/call  Ts/call  name
 0.00      0.00     0.00     1204     0.00     0.00  print_type_debug
 0.00      0.00     0.00      470     0.00     0.00  smalloc
 0.00      0.00     0.00      428     0.00     0.00  sfree
 ......
```

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

4 Evaluation

We will implement an efficient interpreter of lambda calculus by using de Brujin index and closure environment.

the closure environment is recursively defined as

```
val -> Int   /* constant */
    |  Op c  /* primitive operator and c = +, -, *, /, =, < */
    | (ABS T) * env /* val is product of AST with node kind ABS, and E */
env -> [v1; v2; ...; vn]  /* env is a list of value */
```

As we know, the β-reduction:

```
(@x.M) N => M[N/x] /* substitute  each occurrence of x in M by N */
```

with the closure environment, we will replace the heavy jobs of substitutions by the access of environment:

```
<(@x.M) N, env> => <M, N::env> /* N::env is list by add head N in env */
```

if an `x` encountered in the evaluation of `M`, we can get it by the binding deepth in `N::env`, just like get the type of an abstraction variable in typing, or like access the local variable via the **offset**.

4.1 Evaluation of Call-by-value

the syntax-directed reduction rules are:

1. rule of constant

$$\frac{}{\texttt{<CONST n, env> => n}}\ \text{cst}$$

2. rule of variable

$$\frac{}{\texttt{<VAR n, env> => vn}}\ \text{var}$$

where `env=[v1;...;vn;...;vp]`.

3. rule of primitive

$$\frac{\texttt{env=[n; m;...]}}{\texttt{<Op c, n::m::env> => m c n}}\ \text{op}$$

4. rule of abstraction

$$\frac{}{\texttt{<ABS T, env> => (ABS T, env)}}\ \text{abs}$$

5. rule of application

$$\frac{\texttt{<M, env> => (@.M', env')}\quad \texttt{<N, env> => v'}\quad \texttt{<M',v'::env'> => v}}{\texttt{<APP M N, env> => v}}\ \text{app}$$

6. rule of condition

$$\frac{\texttt{<C, env> => n (!= 0)}\quad \texttt{<M, env> => v}}{\texttt{<COND C M N, env> => v}}\ \text{cond}_T \qquad \frac{\texttt{<C, env> => 0}\quad \texttt{<N, env> => v}}{\texttt{<COND C M N, env> => v}}\ \text{cond}_F$$

so `if-then-else` is lazy evaluated

4.1.1 Example 1: `(@x.@y.x y)(@x.x)`

$$\dfrac{\dfrac{}{\texttt{<(@x.(@y.((:2)(:1)))),[]>=>(@x.(@y.((:2)(:1))),[])}}\ \text{abs}\quad \dfrac{}{\texttt{<(@x.(:1)),[]>=>(@x.(:1),[])}}\ \text{abs}\quad \dfrac{}{\texttt{<(@y.((:2)(:1))),[(@.(:1))]>=><(@.((:2)(:1))),[(@.(:1),[])]>}}\ \text{abs}}{\texttt{<(@.(@.((:2)(:1))))(@.(:1))>=><(@.((:2)(:1))),[(@.(:1),[])]>}}\ \text{app}$$

4.1.2 Example 2: `(@x.@y.x y)(@x.x)3`

Let `my=(@x.@y.x y)(@x.x)` in the following reduction tree.

$$\dfrac{\dfrac{}{\texttt{<my, []>=><(@.((:2)(:1))),[(@.(:1),[])]>}}\ \text{ex1}\quad \dfrac{}{\texttt{<3,[]>=>3}}\ \text{cst}\quad \dfrac{\dfrac{}{\texttt{<(:2),[3; (@.(:1),[])]>=>(@.(:1),[])}}\ \text{var}\quad \dfrac{}{\texttt{<(:1),[3; (@.(:1),[])]>=>3}}\ \text{var}\quad \dfrac{}{\texttt{<(:1),[3]>=>3}}\ \text{var}}{\texttt{<((:2)(:1)),[3; (@.(:1),[])]>=>3}}\ \text{app}}{\texttt{<(@x.@y.x y)(@x.x)3, []>=>3}}\ \text{app}$$

4.1.3 Example 3: `+ 3 4`

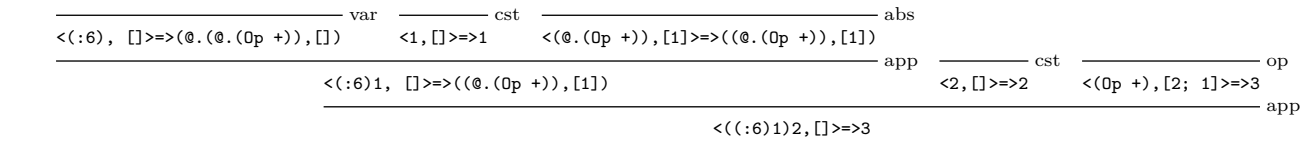
like typing, we divide the closure environment in two parts (static + dynamic), the static parts store pre-evaluated closure of the primitives and predefined terms (`CLOSURE *global_eval_env[MAX_ENV]`). it's initialized as

```
global_eval_env = [(@.@.(Op +),[]); (@.@.(Op -),[]); (@.@.(Op *),[]);
                   (@.@.(Op /),[]); (@.@.(Op =),[]); (@.@.(Op <),[])]
```

we can access the static part with the index `top - n` (`n` is binding deepth), and `top` can bet calculated by `current + length(env)`. for `<(:6), []>` in the following reduction tree, `current` is 6, `length([]) = 0`, so rule var will return `global_eval_env[0]`.

$$\frac{}{\texttt{<(:6), []>=>(@.(@.(Op +)),[])}}\ \text{var}$$

hence

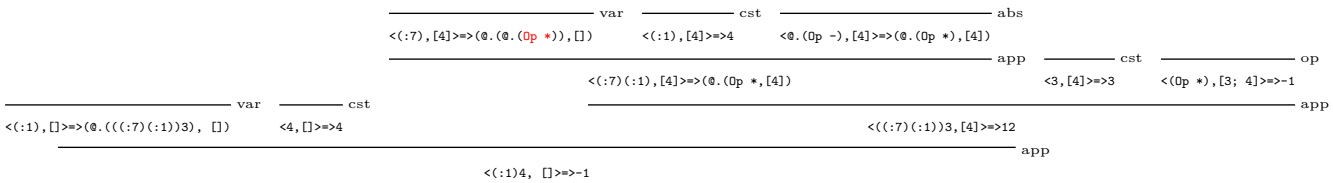


4.1.4 Example 4: `let MY = @x.+ x 3`

firstly, `@x.+ x 3` is evaluated, then store in `global_eval_env[7]`, if `current = 7`, then set `current` to `8`.

$$\frac{}{\texttt{<@x.(((+:7)(x:1))3),[]>=>(@x.(((+:7)(x:1))3),[])}}\ \text{abs}$$

if we input `MY 4;`, the return result is `12`, isn't `7`. what's wrong? let's look the detail of reduction tree:



`current = 8` and `length([]) = 0`, so `global_eval_env[8 - 1]` get perfectly newly stored `MY` pre-evaluated closure.

$$\frac{}{\texttt{<(:1),[]>=>(@.(((:7)(:1))3), [])}}\ \text{var}$$

but in `<(:7),[4]>`, `length([4]) = 1`, so `top = 9` and `global_eval_env[9 - 7]` got `(@.@.(Op *),[])`

$$\frac{}{\texttt{<(:7),[4]>=>(@.(@.(Op *)),[])}}\ \text{var}$$

it's because that we use **wrong** `current` for `MY`. In fact, the environment of `MY` is

```
[4; global_eval_env[5]; global_eval_env[4]; ...; global_eval_env[0]]
```

so `current` for `MY` is `index - 1` where `index` is the index we store `MY` in the global environment `global_eval_env[index]`, so is `6` and hence the real `top` is `6 + 1` and `global_eval_env[7 - 7]` got the correct `Op +`.

so we should store each `current` in the closure environment at name declaration time. like `(@x.(((+:7)(x:1))3),[],7)` to indicate the proper static continuation of environment of `MY` which begins at `global_eval_env[7 - 1]`. After declaration of `MY`, `global_eval_env[]` is

```
global_eval_env = [(@.@.(Op +),[],1); (@.@.(Op -),[],2); (@.@.(Op *),[],3); (@.@.(Op /),[],4);
                   (@.@.(Op =),[],5); (@.@.(Op <),[],6); (@x.(((+:7)(x:1))3),[],7)]
```

And the new reduction rules are:

1. rule of constant

$$\frac{}{\texttt{<CONST n, env, i> => n}}\ \text{cst}$$

2. rule of variable

$$\frac{}{\texttt{<VAR n, env, i> => vn}}\ \text{var}$$

where `env=[v1;...;vn;...;vp]`.

3. rule of primitive

$$\frac{\texttt{env=[n; m;...]}}{\texttt{<Op c, n::m::env, i> => m c n}}\ \text{op}$$

4. rule of abstraction

$$\frac{}{\texttt{<ABS T, env, i> => (ABS T, env, i)}}\ \text{abs}$$

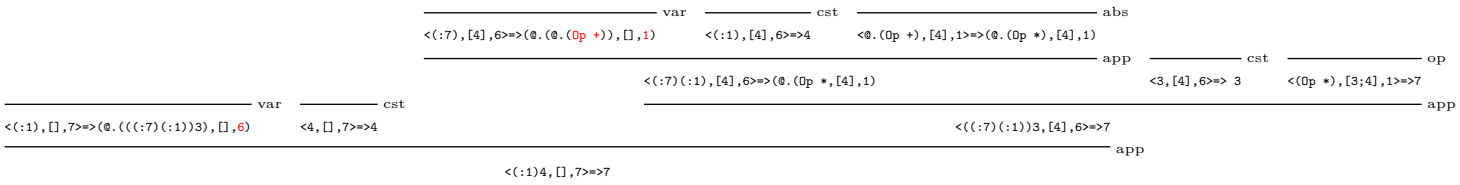
5. rule of application

$$\frac{\texttt{<M, env, i> => (@.M', env', i')} \quad \texttt{<N, env, i> => v} \quad \texttt{<M',v'::env', i'> => v}}{\texttt{<APP M N, env, i> => v}}\ \text{app}$$

6. rule of condition

$$\frac{\texttt{<C, env, i> => n (no 0)} \quad \texttt{<M, env> => v}}{\texttt{<COND C M N, env, i> => v}}\ \text{cond}_T \qquad \frac{\texttt{<C, env, i> => 0} \quad \texttt{<N, env> => v}}{\texttt{<COND C M N, env, i> => v}}\ \text{cond}_F$$

So the correct reduction tree of `MY 4` is



4.1.5 Algorithm

the closure environment is defined as

```
typedef struct Closure {
  AST *ast;  /* use AST for the value */
    /* use VAR 0, VAR -1, ..., VAR -5 represent OP +, OP -, ..., OP <*/
  int index; /* current for access global_eval_env[] */
  struct Closure_list {
    struct Closure *clos;
    struct Closure_list *next;
  } *env;  /* value list for the abstraction value */
} CLOSURE;

typedef struct Closure_list CLOSURE_LIST;
```

and `get_n_th()` in typing is rewrite as

```
CLOSURE *get_global(int i)
{
  return clone_clos( (global_eval_env[i]));
  /* always get the copy of env */
}

CLOSURE *get_argument(int n, CLOSURE_LIST *env, int index)
{
  int i = 0;
  while ( i != n -1 && env != NULL  ) {
    env = env -> next;
```

```
    i ++;
  }

  if (i == n - 1  && env != NULL)  return clone_clos(env -> clos);
    /* always get the copy of env */

  if (index - (n - i) >=0 ) return get_global(index - n + i );
  /* index + i is top of stack is relative top */
  printf("wrong access closure env\n");
  exit (1);
}
```

You should always work with its proper environment with the following duplicate function

```
CLOSURE *clone_clos(CLOSURE *source)
{
  if (source == NULL) return NULL;
  return make_clos(clone_tree(source -> ast),
                   clone_list(source -> env),
                   source -> index);
}

CLOSURE_LIST *clone_list(CLOSURE_LIST *source)
{
  if (source == NULL) return NULL;
  return make_list(clone_clos(source -> clos), clone_list(source -> next));
}
```

so the recursive evaluation of call-by-value

```
CLOSURE *eval_cbv(CLOSURE *clos)
{
  /* always make new return clos and free the evaluated clos */
  AST *exp = clos -> ast;
  CLOSURE_LIST *env = clos -> env;
  CLOSURE *result;
  int index = clos -> index;
  step++;
  switch (exp -> kind) {
  case CONST:
    free_list(env);
    clos -> env = NULL;
    return clos;
  case VAR:
    if (exp -> value <= 0) {
      result = cbv_primitive (clos);
      return result;
    }
    result = get_argument(exp -> value, env, index);
    free_clos(clos);
    return (result);
  case ABS:
    return (clos);
  case COND: {
     /* todo */
  }
  default: { /* APP */
   /* todo */
   /* for (APP M N),
```

```
      1/ eval(M, env, index) to (@.M', env', index')
      2/ eval(N, env, index) to N'
      3/ return eval(M', N'::env, index')
   */
  }
  }
}
```

4.2 Evaluation of Call-by-name

for evaluate `APP(M N)` in normal order evaluation, `N` will delay as **thunk**, a closure like `(N, [])`. But it's question where it is put in the our closure environment? Because it's outmost order, and it's inverse order of the binding deepth! we can't direct put it in the closure of `M`. so we need an extra stack to store the thunk, and for each abstraction, put it back to the closure.

```
   <(((@.@.@.M)N1)N2)N3, [], []> /* third is stack for thunk */
=> <((@.@.@.M)N1)N2, [], [(N3,[])]>
=> <(@.@.@.M)N1, [], [(N2,[]);(N3,[])]>
=> <@.@.@.M, [], [(N1,[]);(N2,[]);(N3,[])]>
=> <@.@.M, [(N1,[])], [(N2,[]);(N3,[])]>
=> <@.M, [(N2,[]);(N1,[])], [(N3,[])]>
=> <M, [(N3,[]);(N2,[]);(N1,[])], []>
```

so we can correct access the closure environment with binding deepth in `M`

the news closure environment for is recursively defined as

```
val -> Int   /* constant */
    |  Op c  /* primitive operator and c = +, -, *, /, =, < */
    |  T * env  /* product of any AST T and env */
env -> [v1; v2; ...; vn]  /* env is a list of value */
stack -> env /* stack for thunk */
```

the reduction of call-by-name is

1. rule of constant

$$\frac{}{\texttt{<CONST n, env, stack> => n}}\ \text{cst}$$

2. rule of variable

$$\frac{\texttt{<tn, en, stack> => v}}{\texttt{<VAR n, env, stack> => v}}\ \text{var}$$

   where `env=[v1;...;(tn,en);...;vp]`.

3. rule of primitive

$$\frac{\texttt{<t1, e1, []> => n}\quad\texttt{<t2, e2, []> => m}}{\texttt{<Op c, (t1,e1)::(t2,e2)::env> => m c n}}\ \text{op}$$

4. rule of abstraction

$$\frac{\texttt{<T, s::env, stack> => v}}{\texttt{<ABS T, env, s::stack> => v}}\ \text{abs}$$

5. rule of application

$$\frac{\texttt{<M, N, (N, env)::stack> => v}}{\texttt{<APP M N, env, stack> => v}}\ \text{app}$$

6. rule of condition

$$\frac{\texttt{<C, env, stack> => n (!= 0)} \quad \texttt{<M, env, stack> => v}}{\texttt{<COND C M N, env, stack> => v}}\ \text{cond}_T$$

$$\frac{\texttt{<C, env, stack> => 0} \quad \texttt{<N, env, stack> => v}}{\texttt{<COND C M N, env, stack> => v}}\ \text{cond}_F$$

so `if-then-else` is lazy evaluated

4.2.1 Example 1: `(@x.@y.x y)(@x.x) 3`

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{<3,[],[]>=>3}}\text{cst}}{\texttt{<(:1),[(3,[]);(@.(:1),[])],[]>=>3}}\text{var}}{\texttt{<(:1),[((:1),[(3,[]);(@.(:1),[])])],[]>=>3}}\text{var}}{\texttt{<@.(:1),[],[((:1),[(3,[]);((@.(:1)),[])])]>=>3}}\text{abs}}{\texttt{<(:2),[(3,[]);(@.(:1),[])],[<(:1),[(3,[]);(@.(:1),[])]>]>=>3}}\text{var}}{\texttt{<(:2)(:1),[(3,[]);(@.(:1),[])],[]>=>3}}\text{app}}{\texttt{<@.((:2)(:1)),[(@.(:1),[])],[(3,[])]>=>3}}\text{abs}}{\texttt{<@.(@.((:2)(:1))),[],[(@.(:1),[]);(3,[])]>=>3}}\text{abs}}{\texttt{<(@.(@.((:2)(:1))))(@.(:1)),[],[(3,[])]>=>3}}\text{app}}{\texttt{<((@.(@.((:2)(:1))))(@.(:1)))3,[],[]>=>3}}\text{app}$$

4.2.2 Example 2: `+ 2 (* 3 4)`

like CBV, we divide the closure environment in two parts (static + dynamic), the static parts store evaluated closure of the primitives and predefined terms (`CLOSURE *global_eval_env[MAX_ENV]`).
it's initialized as same as CBV

```
global_eval_env = [(@.@.(Op +),[]); (@.@.(Op -),[]); (@.@.(Op *),[]);
                   (@.@.(Op /),[]); (@.@.(Op =),[]); (@.@.(Op <),[])]
```

our implementation will use a commnad option (`-v`, choose CBV, default is CBN) to fix the strategie of the evaluation. it can't be changed during the execution of the interpreter.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{<4,[],[]>=>4}}\text{cst} \quad \dfrac{}{\texttt{<3,[],[]>=>3}}\text{cst}}{\texttt{<(Op *),[(4,[]); (3,[])],[]>=>12}}\text{op}}{\texttt{<@.(Op *),[(3,[])],[(4,[])]>=>12}}\text{abs}}{\texttt{<@.(@.(Op *)),[],[(3,[]);(4,[])]>=>12}}\text{abs}}{\texttt{<(:4),[],[(3,[]);(4,[])]>=>12}}\text{var}}{\texttt{<((:4)3),[],[(4,[])]>=>12}}\text{app}}{\texttt{<((:4)3)4,[],[]>=>12}}\text{app} \quad \dfrac{}{\texttt{<2,[],[]>=>2}}\text{cst}}{\texttt{<(Op +),[(((:4)3)4,[]);(2,[])],[]>=>14}}\text{op}}{\texttt{<(Op +),[(2,[])],[(((:4)3)4,[])]>=>14}}\text{app}}{\texttt{<(@.(@.(Op +))),[],[(2,[]);(((:4)3)4,[])]>=>14}}\text{abs}}{\texttt{<(:6),[],[(2,[]);(((:4)3)4,[])]>=>14}}\text{var}}{\texttt{<(:6)2,[],[(((:4)3)4,[])]>=>14}}\text{app}}{\texttt{<((:6)2)(((:4)3)4),[],[]>=>14}}\text{app}$$

4.2.3 Example 3: `let MY = @x.+ x 3`

like CBV, we coded `OP +`, ..., `OP <` with `VAR (:0)`,..., `VAR (-5)`. and add `index` (defined in 4.1.4) in closure. and the new rules are:

1. rule of constant

$$\frac{}{\texttt{<CONST n, env, stack, i> => n}}\ \text{cst}$$

2. rule of variable

$$\frac{\texttt{<tn, en, stack, i'> => v}}{\texttt{<VAR n, env, stack, i> => v}}\ \text{var}$$

   where `env=[v1;...;(tn,en,i');...;vp]`.

3. rule of primitive

$$\frac{\texttt{<t1, e1, i1,> => n} \quad \texttt{<t2, e2, [], i2> => m}}{\texttt{<Op c, (t1,e1,i1)::(t2,e2,i2)::env> => m c n}}\ \text{op}$$

4. rule of abstraction

$$\frac{\texttt{<T, s::env, stack, i> => v}}{\texttt{<ABS T, env, s::stack, i> => v}}\ \text{abs}$$

5. rule of application

$$\frac{\texttt{<M, N, (N,env,i)::stack, i> => v}}{\texttt{<APP M N, env, stack, i> => v}}\ \text{app}$$

6. rule of condition

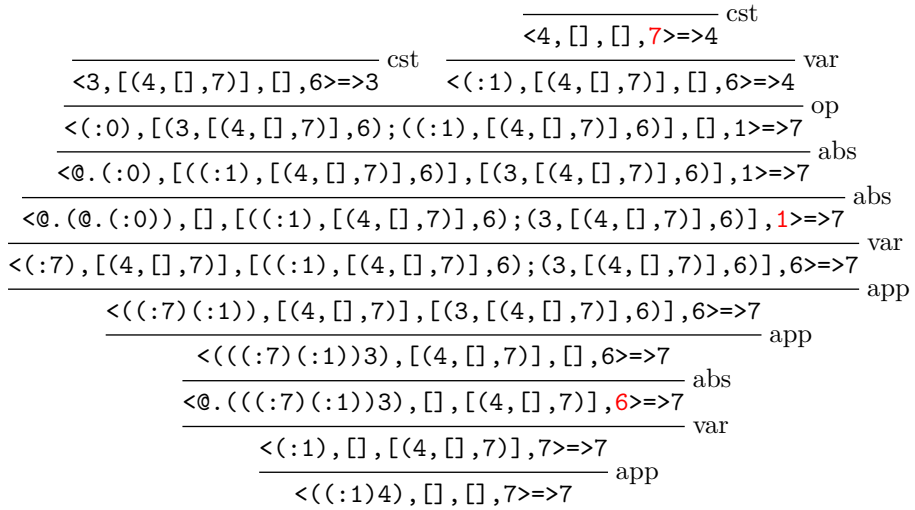
$$\frac{\texttt{<C, env, stack, i> => n (!= 0)} \quad \texttt{<M, env, stack, i> => v}}{\texttt{<COND C M N, env, stack, i> => v}}\ \text{cond}_T$$

$$\frac{\texttt{<C, env, stack, i> => 0} \quad \texttt{<N, env, stack, i> => v}}{\texttt{<COND C M N, env, stack, i> => v}}\ \text{cond}_F$$

for `let MY = @x.+ x 3`, we preevaluate `@x. + x 3`,

`<@.(((:7)(:1))3),[],[],6>=>(@x.(((+:7)(:1))3),[])`

where `VAR (:0)` referes `(OP +)` and `current` is `6`. then store the closure `(@x.(((+:7)(:1))3),[],6)` in `global_eval_env[6]` and increased `current` to `7`. with the `index 6`, we can perfectly access the enviroment of `MY`, if input: `MY 4;`



4.2.4 Example 4: fixed-point combinators

with CBV or CBN, we stop the evaluation if the lambda term which the AST top is an abstraction (much weak than Weak Head Normal Form (https://en.wikipedia.org/wiki/Lambda_calculus_definition#Normal_form)). so for `(@x.(@x.x)x)` isn't evaluated

```
(@x.((@x.(x:1))(x:1))) => (@x.((@x.(x:1))(x:1)))
step = 1
```

and if there is an argument `1` for the above term, both CBV and CBN will return the normal form. for CBV,

```
(@x.((@x.(x:1))(x:1))) => 1
step = 7
```

with those strategies, we can prefined the fixed-point combinators:

```
let Y=@f.(@x.f(x x))(@x.f(x x));
let Z=@f.(@x.f(@y.(x x)y))(@x.f(@y.(x x)y));
```

even with Church's numerial with no lazy `IF` and `Z`

```
let FACT=Z (@f.@n.IF (ISZERO n) ONE (MULT n (f (PRED n))));
```

but in CBV, the following definition will go infinite loop!

```
let FACT=Y (@f.@n.IF (ISZERO n) ONE (MULT n (f (PRED n))));

>>>((:19)(@.(@.((((:64)((:39)(:1)))(:45))(((:36)(:1))((:2)((:40)(:1))))))))
please input a lambda term with ";":

Program received signal SIGSEGV, Segmentation fault.
0xb7e7a7e7 in _int_malloc (av=av@entry=0xb7fb1420 <main_arena>,
    bytes=bytes@entry=20) at malloc.c:3302
```

but it's efficient if use our lazy `if-then-else` and lazy `Z`:

```
let fact = (Z (@f.@n. (if (= n 0) then 1 else (* n (f (- n 1))) fi)));
>>> (fact:1) |== (int -> int)
fact 10;
>>> ((fact:1)10) => 3628800
>>> step = 332
```

with CBV, we can both define:

```
fact 10;
>>> ((fact:1)10) => 3628800
>>> step = 1249

let fact = (Y (@f.@n. (if (= n 0) then 1 else (* n (f (- n 1))) fi)));
fact 10;
>>> ((fact:1)10) => 3628800
>>> step = 1129

let FACT=Y (@f.@n.IF (ISZERO n) ONE (MULT n (f (PRED n))));
>>> ((Y:19)(@f.(@n.((((IF:64)((ISZERO:39)(n:1)))(ONE:45))(((MULT:36)(n:1))((f:2)
>>> ((PRED:40)(n:1)))))))) |== NULL

FACT TWO (@x.+x 1) 0;
>>> ((((FACT:1)(THREE:42))(@x.(((+:77)(x:1))1)))0) |== NULL
>>> ((((FACT:1)(THREE:42))(@x.(((+:77)(x:1))1)))0 => 2
>>> step = 1141

FACT THREE (@x.+x 1) 0;
>>> ((((FACT:1)(THREE:42))(@x.(((+:77)(x:1))1)))0) |== NULL
>>> ((((FACT:1)(THREE:42))(@x.(((+:77)(x:1))1)))0 => 6
>>> step = 6681 /* wait 2 minutes! */

let FACT=Z (@f.@n.IF (ISZERO n) ONE (MULT n (f (PRED n))));
>>> ((Z:20)(@f.(@n.((((IF:66)((ISZERO:41)(n:1)))(ONE:47))(((MULT:38)(n:1))((f:2)
>>> ((PRED:42)(n:1)))))))) |== NULL

FACT TWO (@x.+x 1) 0;
>>> ((((FACT:1)(TWO:45))(@x.(((+:79)(x:1))1)))0) |== NULL
>>> ((((FACT:1)(TWO:45))(@x.(((+:79)(x:1))1)))0) => 2

FACT THREE (@x.+x 1) 0;
>>> ((((FACT:1)(THREE:42))(@x.(((+:77)(x:1))1)))0) |== NULL
killed /* memory exhausted */
```

So the CBN is inefficient, and `Y` is more efficent than `Z` if we code the recursion in CBN. in real world of functional programming language, we can't use this CBN method.

4.2.5 Lazy infinit list

Because the preevaluation of `FIBOGEN ZERO ONE` in CBV will go infinite loop, We add an abstraction in the definition of `FIBO` (see `library.txt`). You should redefine as `let FIBO = FIBOGEN ZERO ONE` in CBN mode. `GETN NUM FIBO (@x.+x 1) 0` will retrieve the `NUM`-th element of the infinite Fibonacci sequence.

```
let FIBO=FIBOGEN ZERO ONE;
GETN THREE FIBO (@x.+x 1) 0;
>>>(((((GETN:14)(THREE:42))(FIBO:1))(@x.(((+:77)(x:1))1)))0) => 2
step = 1475
```

4.3 Memory Leaks

We can gprofile the memory leaks:

1. CBN

   ```
   $ ./lambda
   SUM TWO (@x.+ x 1) 0;
   >>> 3
   ```

```
>>> step = 611
CTRL+D
$ gprof ./lambda
%   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
31.25      0.05     0.05   356729     0.00     0.00  free_ast
15.62      0.07     0.03   356729     0.00     0.00  free_clos
12.50      0.10     0.02  4309502     0.00     0.00  smalloc
12.50      0.12     0.02  1398179     0.00     0.00  make_app
12.50      0.14     0.02   356747     0.00     0.00  free_list
 6.25      0.14     0.01  3588583     0.00     0.00  make_ast
 6.25      0.15     0.01   356542     0.00     0.00  clone_tree
 3.12      0.16     0.01   356781     0.00     0.00  clone_list
 0.00      0.16     0.00  4306063     0.00     0.00  sfree
......

$ ./lambda
SUM THREE (@x.+ x 1) 0;
>>> 6
>>> step = 1390
CTRL+D
$ gprof ./lambda
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
24.83      0.73     0.73  6772543     0.00     0.00  clone_tree
19.05      1.29     0.56  6772966     0.00     0.00  free_ast
 9.52      1.57     0.28 69207383     0.00     0.00  make_ast
 7.82      1.80     0.23  6772999     0.00     0.00  free_list
 6.80      2.00     0.20  6773015     0.00     0.00  clone_list
 5.44      2.16     0.16 82760466     0.00     0.00  smalloc
 5.44      2.32     0.16 27044285     0.00     0.00  make_app
 5.27      2.48     0.15  6772543     0.00     0.00  clone_clos
 4.42      2.60     0.13 82757027     0.00     0.00  sfree
......
```

so `smalloc - sfree` is constant

2. CBV

```
$ ./lambda -v
fact 5;
>>> 120
>>> step = 172
CTRL+D
$ gprof ./lambda
   %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
 0.00      0.00     0.00    25555     0.00     0.00  smalloc
 0.00      0.00     0.00    21119     0.00     0.00  sfree
 0.00      0.00     0.00    15718     0.00     0.00  make_ast
......

$ ./lambda -v
fact 10;
>>> 3628800
>>> step = 332
CTRL+D
$ gprof ./lambda
```

```
  %   cumulative   self              self     total
time   seconds   seconds    calls  ms/call  ms/call  name
 0.00      0.00     0.00    33665     0.00     0.00  smalloc
 0.00      0.00     0.00    29229     0.00     0.00  sfree
 0.00      0.00     0.00    22448     0.00     0.00  make_ast
 0.00      0.00     0.00     8670     0.00     0.00  make_var
......
```

so `smalloc - sfree` is also constant

5 TODO

Complete `eval_cbv()` and `eval_cbn()` of `closure.c`.

please send your `closure.c` as attached file to `mailto:hfwang@whu.edu.cn?subject=ID(06)` where the `ID` is your student id number.

–hfwang December 18, 2017