TLC (Tiny Lambda Calculus) 语言编译器的设计与实现

WANG Hanfei

March 20, 2022

Contents

2019 级弘毅班《编译原理》第 4~7 次课程实验

1 简介

本次实验目标是实现最简函数式程序设计语言 TLC (Tiny Lambda Calculus) 的编译器，该编译器以闭包 (closure) 作为运行环境.

λ 演算是以变量绑定 (variable binding) 和代入 (substitution) 来表达计算. 它既可以作为计算模型，也可以作为逻辑模型 (see https://en.wikipedia.org/wiki/Lambda_calculus).

它是函数式程序设计语言的底层模型 (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 TLC 文法

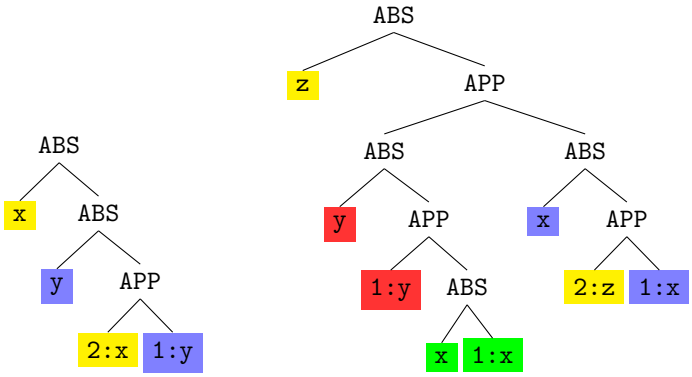
TLC 的文法如下:

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

其中:

1. expr 所定义的成分称为 λ 项.
2. @x.M 称为抽象 (abstraction, 由于键盘无法直接输入"λ", 我们用 @ 替代), 其中 M 称为抽象体 (*abstraction body*). x 称为抽象名.
3. M N 称为应用 (application)，即函数 M 作用实参 N.
4. if-then-else 结构的加入主要是为了条件表达式的惰性求值.
5. let X = M 称为 *let-binding*, X 称为 let-binding 名. 它在全局环境中绑定变量 X 为 λ 项 M. 这样在下文中自由出现的 X 即是 M.
6. 应用是左结合的，即 M N P $\equiv$ (M N) P.
7. 运算的优先级由低到高：if-then-else, 抽象，应用

(see lexer.l and grammar.y in detail).



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

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

1.2 抽象语法树 (Abstract syntax tree)

我们在语法分析时同步构造 TLC 语言的抽象语法树. 其中约束变量用 De Brujin index 表示, 即在抽象体中出现的 `x` 被多少个抽象所嵌套 (也称为 *binding depth*).

如 `@x.@y.x` 中的抽象体中出现的 `x` 被从其定义点 `@x` 开始有两个抽象 (`@x.@y.`), 故其 binding depth 为 2. 即 `@x.@y.2`. 而 `@z.(@y.y(@x.x))(@x.z x)` AST 为 `@z.(@y.1(@x.1))(@x.2 1)` (see Figure 1).

如同栈式的运行环境的局部变量的 offset, Binding depth 是访问 closure 运行环境的钥匙.

由于抽象体的变量一定有 binding depth, 即被某个抽象所约束，因此 TLC 语言不允许有自由变量出现.

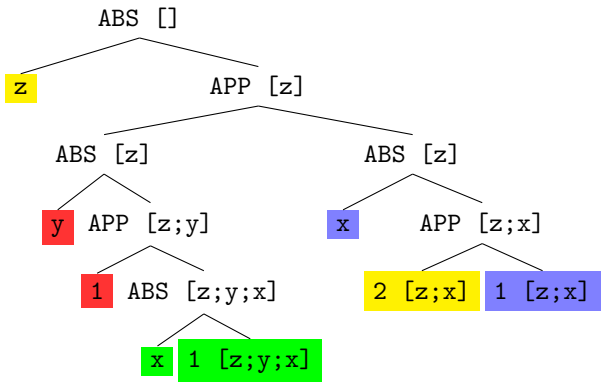
TLC 语言的 AST 定义如下：

```
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 Depth 的计算

AST 结构中除了 binding depth，其他的都是综合属性，可用 YACC 产生式尾部的语义动作完成计算 (见 `grammar.y`). 我们用一个静态栈 `char *name_env[MAX_ENV]` 记录抽象层次, 用 `int current` 来跟踪下一个可用的栈顶，即当前栈顶为 `current - 1` (`tree.c`). 在构造 AST 时，每次遇见 `ABS` 结点, 压抽象名入栈, 离开 `ABS` 时 pop 栈. 见如下 `grammar.y` 代码片段：

```
/* in grammar.y */

expr : INT
| ID {
   int depth = find_depth((char *) $1 -> lchild);
   $$ = $1, $$ -> value = depth;
 }
```



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

Figure 2: Binding depth

```
...
| '@' ID { /* TODO*
    /* midaction, push stack ID on name_env[] stack */
   } '.'  expr %prec THEN {
     /* TODO */
    }
...
;
```

当在抽象体中遇见 `ID` 时，`find_depth(char *name)` 将返回当前 `ID` 名在栈中的深度，即 `binding depth`:

```
int find_depth(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);
}
```

Figure 2 的 AST 树的结点附注 (右边的链表为 `name_env[]`, 栈顶在右边) 反应了栈的变化情况，叶节点的数字即是 binding depth.

1.4 原始运算和 let-bindings 的处理

不同与命令式程序设计语言，函数必须有名. λ 项中函数作为与运算量同等的一阶公民必须匿名, let-binding 在全局或在局部对 λ 项命名实际上是 syntactic sugar. 如语句`"let I = @x.x; I 3;"` 等价于`"(@I. I 3)(@x.x);"`. 即 `I` 是可看成抽象名, `(@x.x)` 可看成是 `I` 的实参. 所以当语法分析 `let I = @x.x` 时，压 `I` 入 `name_env[]` 栈. `I` 所定义的 λ 项 `@x.x` 作为函数作用的实参来处理 (见3. Evaluation). 故当分析 `I 3` 时，其 `I` 的 binding depth 为 `1`.

多个 let-binding, 也是 sugar. 如:

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

这时的 `I 3` 可看成 `(@I.(@K.I 3)(@x.@y))(@x.x)`, 所以 `I` 的 binding-depth 为 `2`.

对此我们采用的方法是让 let-binding 名保存 `name_env` 中，作为全局的环境，当下文中再次出现时，`find_depth()` 根据当前的栈顶获得新的 binding depth. 这样，语法分析在第一个 `let` 之后，`name_env[] = [I]`, `current = 1`; 第二个 `let` 之后，`name_env[] = [I,K]`, `current = 2`. 再分析 `I 3`, 调 `find_depth()` 返回 binding depth 为 2, 即 `(I:2) 3`, 与 sugar 一致. 所以同一个 let-binding 名，在程序中的不同位置，其 binding depth 可能不同.

算术运算可看成是预定义的 let-binding 名. 所以 `name_env[]` 初始化为 (see `grammar.y`):

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

由于 λ 演算只能定义一元函数, 多元函数必须柯里化为高阶函数. 因此上述原始运算应解释为 `@x.@y.op x y`. 这样中缀的算术表达式 `2 * 3 + 4` 在 TLC 中应写成 `+ (* 2 3) 4`. 注意不是前缀表达式! 如前缀表达式 `+ * 2 3 4` 能正确地语法分析为: `(((+ *) 2) 3) 4`, 但有语义错误: `+` 只能作用整型量，而不是函数 `*`.

在命令行运行"`./lambda`"，输入:

```
+ (* 2 3) 4;
```

语法分析为 `(((+:6)(((*:4)2)3))4)` 其中 6 和 4 分别是 `+` 和 `*` 的 binding depth.

继续输入：

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

语法分析保存 `I` 到 `name_env[current]`. 这时

```
name_env[] = {"+", "-", "*", "/", "=", "<", "I"}
current = 7
```

继续用 let-binding 定义 `PLUS`:

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

当语法分析到抽象体 `+ x y` 时，`@x` 和 `@y` 已压栈：

```
name_env[MAX_ENV] = {"+", "-", "*", "/", "=", "<", "I", "x", "y"}
current = 9
```

这时 `find_depth("+")` 返回 9, `find_depth("x") = 2`, `find_depth("y") = 1`.

故 `@x.@y.+ x y` 的 AST 为 `(@x.(@y.(((+:9)(x:2))(y:1))))` (见 Figure 3).

当分析完上 let-binding 语句后:

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

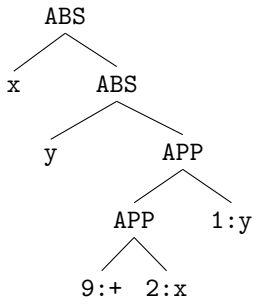
若继续定义 `PLUS2`:

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

`@x.@y.+ x 2` 的 AST 为 `(@x.(@y.(((+:10)(x:2))(y:1))))`. 注意到 `+` 的 binding depth 变为 10 (见 Figure 4). 这是因为定义 `PLUS` 后 `name_env[]` 增加了 `PLUS`, 这样 `+` 相对于栈顶的距离也增 1.

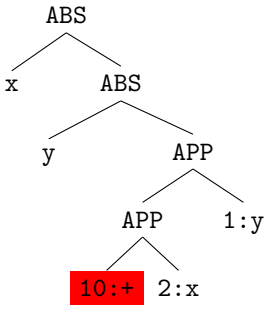
定义 `PLUS2` 之后, `name_env[]` 变为:

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



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

Figure 3: AST of `PLUS`



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

Figure 4: AST of `PLUS2`

1.5 "=" 的处理

由于运算符号 ("+", "-", ..., "=") 被看 let-binding 的 `ID`, 因此词法分析时，运算符号识别为单词 `ID` 即可.

但由于"=" 在 let-binding 中要求词法返回单词 '='.
我们可利用语法分析的上文信息来判断 `=` 识别为 `ID` 还是 '=', 即设置全局变量 `int is_decl` (见 grammar.y), 缺省设为 0, 表示 `=` 作为正常的 `ID`. 当上文出现 `LET` 时，置 `is_decl` 为 1 表示要 `=` 是 let-binding 中出现的正常等号, 返回 '='. 可使用 YACC 的中部语义动作来激活 `is_decl`:

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

词法分析在返回'=' 时，对 `is_decl` 置零 (见 lexer.l):

```
"=" {
      char *id;
      if (is_decl) {is_decl = 0; return '='; }
      id = strdup(yytext);
      yylval = make_string(id);
      return ID;
}
```

1.6 输出 AST

我们使用 LaTeX 作图包 tikz/pgf (https://sourceforge.net/projects/pgf/) 和 tikz-qtree (https://ctan.org/pkg/tikz-qtree) 输出 AST 树结构. 函数 `printtree(AST *)` 输出 LaTeX 源文件 `expr.tex` 它是模板 exptree.tex 的 included 文件. 这样"`pdflatex exptree.tex`" 即可转换 AST 为图形 exptree.pdf.

1.7 TODO

完成 `grammar.y`, 使得程序能输出与样本程序一样的 AST. 请用 `library.txt` 测试你的程序.

请将你的 `grammar.y` 作为附件 `mailto:595180978@qq.com?subject=ID(04)` 其中 `ID` 为你的学号.

DDL：待定.

~

–hfwang March 20, 2022

2 类型

> Well-typed programs cannot go wrong — Robin Milner

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

2.1 语法制导类型合成

λ 演算把程序和数据都统一为 λ 项，可实现程序的高度抽象。但也带来了风险。如文法正确的 λ 项`"x x"` 可导致罗素悖论 (见 L. Paulson's lecture " Foundation of Functional Programming"，PP. 23). 为了避免这样的情况发生，我们对每个 λ 项进行定型 (typing)，不能定型的 λ 项将被系统拒绝.

如 `x x` 中的第一个`"x"` 必须是函数，才能作用实参，不失一般性可假定第一个 `x` 的类型是`"A -> B"` (`A` 到 `B` 的函数集合, `A` 和 `B` 是任意的集合, 称之为 类型变量 *type variables*). 这样作为实参的第二个`"x"` 必须是 `A` 类型的变量 (see `https://en.wikipedia.org/wiki/Simply_typed_lambda_calculus` or Pierce's Book "Types and Programming Languages", Ch. 9: Simply Typed Lambda-Calculus). 由于前后两个 `x` 是同一个 λ 项，因此是相同的类型，即

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

我们希望该方程有解. 但为了解方程，首先需要定义在怎样的集合上求解.

为此归纳定义类型集合：

1. `int` 是类型，称为类型常量.
2. `X`, `Y`, `Z`, ..., 字母表中的字母是类型，称为类型变量.
3. 若 `A` 和 `B` 是类型, 则 `A->B` 是类型，称为箭头类型 (*arrow type*). 表示定义域为 `A`, 值域为 `B` 的函数. 如：`X->X`, `X->int`, `(X->Y)->Z` 等.

上述方程中由于等式左边的 `A` 在等式右边作为箭头类型的定义域出现，因此类型变量 `A` 和 `B` 在类型集合取任元素都不能使得等式相等，即无解，也称 λ 项不可配型 (*untypable*).

但 λ 项 `(@x.x)(@x.x)` 可配型为 `X->X`. 该项中的第一个 `(@x.x)` (记为 `alpha`) 可配型为 `(A->A)`; 第二个 `(@x.x)` (记为 `beta`) 可配型为 `(B->B)`. 而项`"alpha beta"` 有意义, `alpha` 的定义域 `A` 必须与 `beta` 的类型一致，即满足下述类型方程:

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

这样 `A = C->C` 且 `B = C` 是上述类型方程的一个解.

所以 `alpha` 可配型为 `(C->C)->(C->C)`, `(@x.x)(@x.x)` 的类型是 `alpha` 的值域类型，即 `C->C`. 注意：类型变量 `C` 可以是任意的具体类型，如 `int`, `int->int` 等.
上述方程实际上有无数解：

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

但是所有的这些解都是把第一个解的 C 用某个类型替换得到. 如 C 用 D->D 替换得解 A = (D->D)->(D->D) 且 B = D->D. 我们称第一个解为通解 (*most general*).

上述两项的区别在于 x 在两项中的出现的不同，(x x) 中 x 必须是同一个 x, 而 (@x.x)(@x.x) 中的 x 是不同的 x.

本次实验的目标就是用语法制导类型合成的方法 (*syntax-directed type synthesis*) 给输入的 λ 项配型 (typing), 即若存在，输出其通解；若不存在报类型错误.

2.2 类型合成的单步分解

类型表达式的数据结构定义如下 (见 type.h):

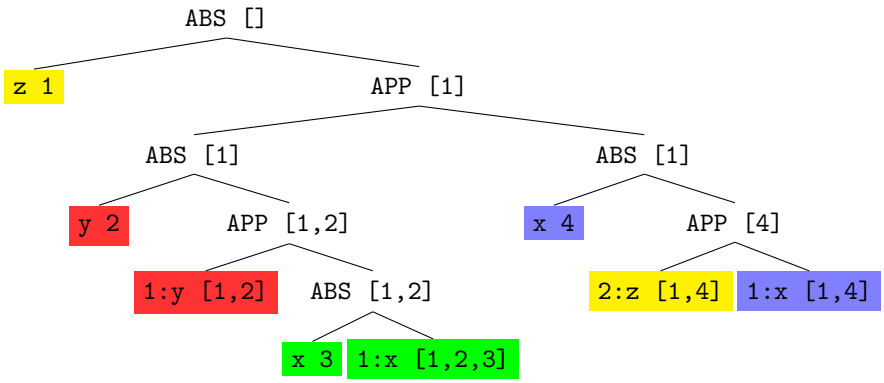
```
typedef enum { Typevar = 1,  Arrow = 2, Int = 3 } Type_kind;
  /* for type tree node */

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[], global_type_env[]
     will store the type for the declared lambda term */
```

其中:

1. 用自然数增序对每次创建的类型表达式 (类型变量或箭头类型) 进行编码 (see make_vartype() and make_arrowtype() in type.c).
2. 保存每次创建的类型表达式到全局环境 Type_env type_env[MAXNODE] (see type.c), 且编码为 i 的类型表达式保存在 type_env[i].
3. 在对 λ 项配型过程中，需通过binding depth获得约束变量的类型. 为此如同 name_env[] 获得 binding depth 一样, 用一个类型栈保存抽象名的类型. 在对 *lambda* 项的 AST 先序遍历时，遇见 ABS 结点，创建一个新的类型变量并类型压栈. 结束抽象体后出栈. 当后序遍历抽象体时，遇到约束变量即可通过其 binding depth 在类型栈获得其类型. 如 @z.(@y.y(@x.x))(@x.z x) (见 Figure 5), 先序遍历为每个抽象名创建类型 (编码从 1 到 4). 其对应的的类型栈如 Figure 5所示. 在子项 @x.x 中的 x 其 binding depth 为 1, 即可在其对应的类型栈 [1,2,3] 的栈顶获得其对应的类型编码 3. 而子项 @x.z x 的 z 的 binding depth 为 2, 在其对应的类型栈 [1,4] 的栈顶下的第二个位子可取到 z 的类型编码 1

   与 name_env[] 一样, 类型栈可设计为用静态数组. 但为了预热下一个设计 (*lambda* 演算的求值), 我们把类型栈分拆成两个部分：静态栈 + 动态栈. 静态栈 global_type_env[] 保存 let-binding 名的类型，动态栈 Var_list abs 保存当前 *lambda* 项的抽象名的类型. 即 abs 是 global_type_env[] 的延伸. 如何用 binding depth 访问类型栈请参见 Example 5

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



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

Figure 5: Binding depth for retrieve the type

Notation: 类型 `type_env[n]` 记为 `n(redirect, type)`, 其中 `n` 类型编码, `redirect` is 或为 `n` (即无重定向) 或为 重定向对应的类型编码. `type` 或为 `n` (即类型变量), 或为 `int` (类型常量), 或为 `(x->y)`. 如

`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)]`

其中 `3(2,3)` 表示编码为 3 的类型重定向到编码为 2 的类型.

记 `M |== T` 为 λ 项 `M` 有类型通解 `T`.

全局环境初始化为:

```
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 */
```

类型常量 `int` 的编码为 0. 类型变量编码从 1 开始.

对每个 λ 项的配型 `type_env[]` 都初始化为:

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

初始化通过下函数完成:

```
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;
}
```

下面一组例子分解对 λ 项 AST 的递归遍历合成其类型的过程.

2.2.1 Example 1: @x.@y.x y (MN: 其中 M 是类型变量)

设 AST 的前序遍历已将抽象名对应的类型压 abs 栈. 例子单步化仅后序遍历合成最后的类型.

step 1

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

setp 2

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

step 3

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

项 x y 有意义, x 必须是箭头类型. 但此时 x 的类型是类型变量 1. 我们可通过函数 get_instance(Type_ptr) 新产生两个类型变量 3 和 4 并把 1 的类型替换为箭头类型 3->4:

```
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;
}
```

它修改通过新生成两个类型变量 3 和 4, 将 1 的类型修改为 3->4. 而 y 作为实参，其类型必须与 x 的定义域类型, 即 3 一致 (*unified*). 一个简单的做法是直接将有类型 3 出现的地方全都修个为 2. 但这样做代价非常大，即需要查找整个 type_env[], 凡是有 3 的出现均修改为 2. 这相当于在数据库中修改主键的键值将导致以该主键为外键的所有记录必须都做相应的修改一样, 代价太大！为了简化这项繁琐的操作，我们只需将类型变量 3 定向到 2 即可 (把 type_env[3] 的 redirect 置为 2). 这项工作由有“副作用”的函数 unify_leaf() (其副作用是改变了 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;
}
```

type_env[] 的类型变量 n(m, t) 称为终止的 (*final*) 当且仅当它没有重定向，即 n == m. 函数 int final_type() 可通过重定向链获得其终止类型. 注意在类型合一过程中所用到的类型必须是终止类型.

```
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 的类型是 2, x y 的类型是 4, 这样抽象 @y.x y 的类型是 2->4. 为此新产生的 type_env[5] 上保存该箭头类型.

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))
```

x 的类型是 3->4, @y.((x:2)(y:1))) 的类型是 2->4, 这样 @x.(@y.((x:2)(y:1))) 的类型是 ((3->4)->(2-> 4)), 在新产生的 type_env[6] 上保存该箭头类型.

step 6

printtype(Type_ptr) 把类型变量用大写字母 A - Z 表示并打印输出, 如 2 用 A, 4 用 B 表示:

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

step 7

配型完成后，释放所有的动态内存. 由于所有创建的类型都记录在 type_env[] 中，因此遍历 type_env[] 即可全部释放.

```
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` (`MN`: `M` 是箭头类型, `N` 是类型变量)

前 5 步同 Example 1. 由于 `(@x.@y.x y)` 是第一个 `@x` 的子项，线序遍历第一个 `@x` 时配型为类型变量 `1`, `(@x.@y.x y)` 的配型从 `2` 开始，故在 Example 1 的所有类型编码上加上 `1` 即可:

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)
```

此时 `((@x.(@y.((x:2)(y:1))))` 已是箭头类型 `((4->5)->(3->5))`, 其中 `(4->5)` 的编码为 `2`, 实参 `x` 为类型变量 `1`. 类型 `4->5` 和 `1` 的合一只需 `1` 定向到 `4->5` 的类型编码，即 `2` 上即可:

```
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))
```

注意到类型 `8(8,(1->(3->5)))` 的定义域类型 `1` 不是终止的, 其终止类型为 `2(2,(3->5))`. 这样

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) (MN: M and N are both arrow)

前 5 步同 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)
```

这时实参 @x.x 的类型为 (7->7), 需与函数的定义域类型 (2->4) 合一. 即 2 与 7 及 4 与 7 需合一, 而 unify(2, 7) 是 unify_leaf(), 即 2 定向到 7 即可. 同样 unify(4, 7) 也是 4 定向到 7 .

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!
```

第一个 x 是类型变量 1, 必须是箭头类型. get_instance(1) 重写它为 2->3. 配型的副作用同时也改变了作为实参的相同的 x. unify(2, 1) 即是 unify_leaf_arrow(2, 1). 若定向 2 到 1, 而 1 的类型是 2->3, 定向链产生了循环. final_type() 会死循环. 因此必须严格禁止. is_occur_node(2, (2->3)) 检查是否有循环定向的发生，若发生，返回 NULL, 即不可配型.

```
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;` (let-binding)

由于 let-binding 实际上是抽象和应用的 syntactic sugar. 如同 AST 的 name_env[], 我们用一个与 name_env 平行的静态栈 global_type_env[] 保存 let-binding 名 MY 的类型. 但需重新整理类型编码.

前 5 步同 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

如同语法分析时，let-binding 名 MY 保存到全局栈 name_env[6], 其类型保存到 global_type_env[6]. 但为了后续引用 MY 的类型时，不破坏配型环境 type_env[], 我们需对 (3->4)->(2->4) 消除定向，并重新从 1 到 n 进行顺序编码 (编码 0 始终保留给 int):

```
/* 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;
    }
}
```

这样 `global_type_env[6] = (1->2)->(1->2)`, `currrent` 加一到 7.

2.2.6 Example 6: `@x.MY x`

本质上是与Example 2相同的项，但子项 `MY` 是 let-binding 名. 因此需要用其 `binding-depth` 正确地访问到 `global_type_env[6]` 保存的类型，并重新编码到当前的 `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` 的 binding depth 为 2, 比 `abs` 栈要大,因此应保存在 `glolal_type_env[]` 中, 而 `glolal_type_env[0..6] + abs` 是完整的类型栈, 其中栈顶在右. 这样从栈顶下的第二个位置 (`top - binding-depth`)，即 `global_type_env[6]` 可取到 `MY` 的类型.
`get_n_th(abs, 2, 8)` 获取 `MY` 的类型，并调用 `restoretype()` 在当前的 `type_env[]` 下重新从 2 到 6 编码为 `6(6,((3->2)->(3->2)))`.

```
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 depth of the lambda variable */
```

```
Type_ptr get_n_th(Var_list_ptr list, int n, int pos)
{
  int i = 0;

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

  while (i != n - 1 && list != NULL) {
    list = list->next;
    i++;
  }

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


  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: 递归

不动点算子 (https://en.wikipedia.org/wiki/Fixed-point_combinator) 可定义为:

```
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));
```

由于 `x x` 不能配型，所以当输入不动点算子时，报类型错误:

```
@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
```

若把不动点算子的类型设为 `(A->A)->A`, 则:

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

则可正确地配型阶乘函数

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

可让 `get_n_th_from_global(int i)` 在 `name_env[top - i]` 中查看当前 let-binding 名是否为 `Y`, `Z`, 或 `rec`. 若是直接返回类型 `(A->A)->A` 即可.

2.3 Church 编码和类型

Church 编码是把数据和运算用 λ 项表示 (https://en.wikipedia.org/wiki/Church_encoding), 详见 library.txt. 请用此文件测试配型算法.

1. Church 数:

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

2. 算术运算:

   ```
   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. 布尔量

   ```
   TRUE |== (A->(B->A))
   FALSE |== (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
```

这样 `EQ` 和 `EQ1` 不能配型!

4. 递归

```
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)
```

可见 `FACT` 和 `SUM` 不能配型, 甚至加上公里 `Y |== (A->A)->A`, 也不行. 但 Ackermann 函数 (https://en.wikipedia.org/wiki/Ackermann_function) 可配型.

2.4 TODO

2.4.1 合一算法

参考龙书合一算法 (PP. 397), 完成下述通过副作用实现合一的函数 `unify()`:

```
/* 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;
}
```

2.4.2 类型合成的语法制导语义定义

树 AST T 有下述三属性:

1. T.top : = current + abstraction depth
2. T.abs: 抽象名对应的类型栈.
3. T.type: 类型.

此外还有全局量: 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 */

2.4.3 配型

属性 T.abs and T.top 是 L 属性，而 T.type 是综合属性. 因此可通过对 AST 递归遍历实现属性计算. 请完成下述递归遍历配型函数:

```
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;
}
```

2.4.4 内存泄露

程序要求一定没有丝毫的内存泄露!!! 可通过输入一个或多个相同的 *lambda* 项:

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

`ctrl+d` 结束程序，运行 `gprof ./lambda`. 其中 `smalloc - sfree` 应是相同的, 如:

输入一次上述项, `profile`:

```
 %   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
 ......
```

输入两次上述项, `profile`:

```
 %   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
 ......
```

则 `268 - 226 = 470 - 428`.

please send your `type.c` as attached file to `mailto:595180978@qq.com?subject=ID(05)` where the `ID` is your student id number.

–hfwang March 20, 2022

3 求值

我们将利用 binding depth (De Brujin index) 和闭包实现一个高效的 λ 演算的解释器.

闭包 (closure) 在上世纪 60 年代由 Peter J. Landin 在实现函数式语言 `ISWIM` 提出 (see The Next 700 Programming Languages, Communications of the ACM in 1966, 请参考 Abstract Machines).

闭包作为运行环境支持局部函数，函数作为参数和返回值, 它可定义为有值和运行环境组成的二重组 `<val, env>`, 其归纳定义如下:

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

其中 `(ABS T)` 是函数, `env` 是该函数的运行环境, 而闭包本身也可作为值而构成嵌套的运行环境.

λ 演算的求值可仅用一条规则 β 归约描述，即:

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

此过程非常繁琐，需要把 `M` 中的每个自由出现的 `x` 替换成 `N`, 且变量的约束与被约束的关系不能改变，若直接对其编程，效率非常低 (see `./Lambda.v` in detail). 闭包环境类似惰性求值，把 `N` 压入 `M` 的运行环境 `env` 中，当对 `M` 求值时遇见 `x`, 即可通过其 binding-depth, 在 `N::env` 中获取到 `N`:

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

该方法与配型时通过 binding depth 在栈 `global_type_env[] + abs` 中取抽象名的类型一样. 也与栈式的运行环境通过 offset 访问局部变量类似.

3.1 Call-By-Value (CBV) 求值

the syntax-directed CBV 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]`.

   let-binding's name will follow this rule.

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

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

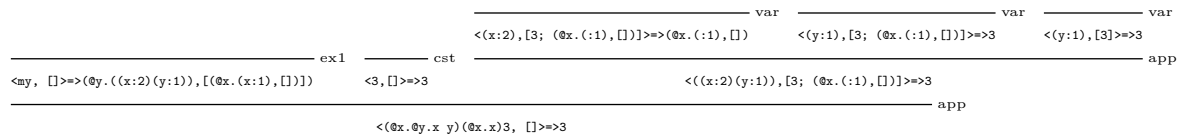
   so `if-then-else` is lazy evaluated

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

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

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

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



3.1.3 Example 3: `+ 3 4`

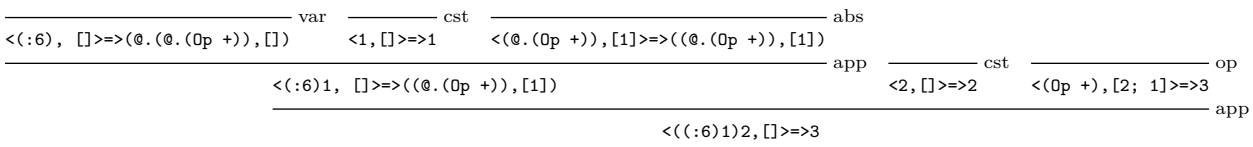
like typing, we divide the closure environment in two parts `global_eval_env[] + env` (static + dynamic), the static parts store pre-evaluated closure of the primitives and let-binding's name (`CLOSURE *global_eval_env[MAX_ENV]`). it's initialized as (because we use only the binding depth in the evaluation, we will write `@x.@y.(x:2)(y:1)` simply as `@.@.(:2)(:1)`)

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

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

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

hence



3.1.4 Example 4: `let MY = @x.+ x 3` (let-binding's name)

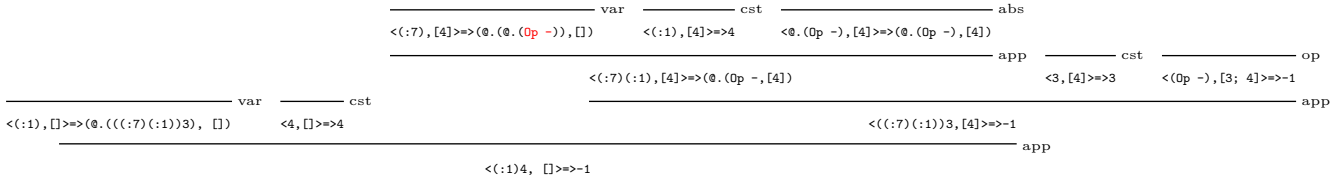
firstly, `@x.+ x 3` is evaluated, and store the value in `global_eval_env[current]`. where the next available place for global envrionments `current = 6`, then increase `current` to 7.

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

so

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

if we input `MY 4;`, the return result is `-1`, isn't 7. what's wrong? let's look the detail of reduction tree:



for `<(MY:1)4, []>`, `current = 7` and `length([]) = 0`, so `global_eval_env[7 - 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 = 8` and `global_eval_env[8 - 7]` got `(@.@.(Op -),[])`, not the expected `Op +`:

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

it's because the real closure for `<(:7),[4]>` is the environment of `MY` in parsing time pushed by `4`

```
bottom of stack                                                          top
------------------------------------------------------------------------------
[(@.@.(Op +),[]);    [(@.@.(Op -),[]);  ...;  (@.@.(Op <),[]);    4
 ------------------  ------------------       ----------------    -
 global_eval_env[0]; global_eval_env[1]       global_eval_env[5]; env[0]
```

but we use the actual environment:

```
bottom of stack                                                          top
------------------------------------------------------------------------------
[(@.@.(Op +),[]);    [(@.@.(Op -),[]); ...; (@.@.(Op <),[]);  (@x.(((+:7)(x:1))3),[]);  4]
 ------------------  ------------------     ----------------  ------------------------ --
 global_eval_env[0]; global_eval_env[1]   global_eval_env[5]; global_eval_env[6];   env[0]
```

the only difference of the above 2 closures is the static env. in other words `current` for `MY` is not the actual 7, is the 6 in the parsing time of `MY`.

hence for correct access the static env of let-binding's name, the `current` in parsing time of let-binding's name must be an argument of the closure. the modified closure is the triple `(M, env, current)` where `current` is actual `current` of `M`.

So our new `global_eval_env[]` after the declaration of `MY` 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

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

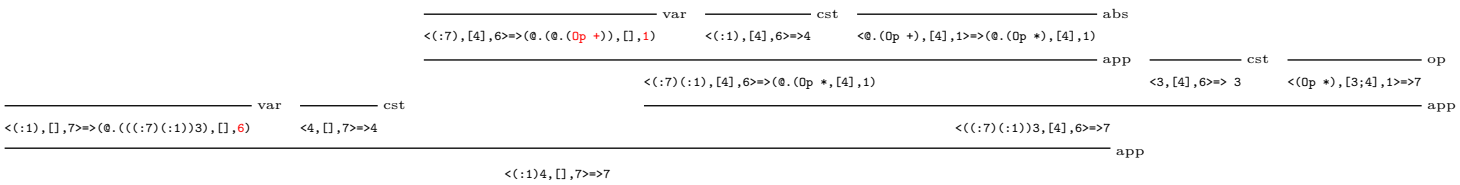
5. rule of application

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

6. rule of condition

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

So the correct reduction tree of `MY 4` is



3.1.5 算法

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 let-binding's name */
  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 rewritten 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;

  if (index - (n - i) >=0 ) return get_global(index - n + i );

  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 */

  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')
   */
  }
  }
}
```

3.2 Call-By-Name (CBN) 求值

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 normal order evaluation is outmost order, and it's inverse order of the binding depth! 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 depth in `M`.

the new closure environment for CBN 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 */
```

and we denote the evaluation of CBN as:

```
<M, env, stack> => v
```

the syntax-directed CBN reduction rules are:

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

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

4. rule of abstraction

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

5. rule of application

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

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

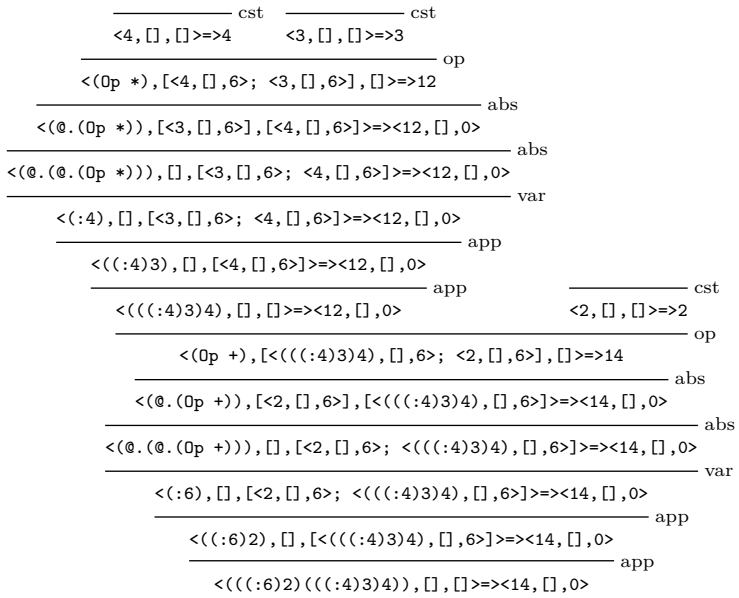
3.2.1 Example 1: `(@x.@y.x y)(@x.x) 3`

3.2.2 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 let-binding's names (`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 (`./lambda -v`, for CBV, default is CBN) to fix the strategie of the evaluation. it can't be changed during the execution of the interpreter.

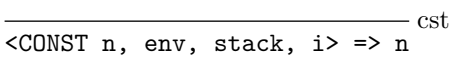


3.2.3 Example 3: `let MY = @x.+ x 3` (let-binding's name)

like CBV, we add `current` of parsing time in closure for let-binding's name (see 3.1.4). and the closure becomes quadruple `<M, env, stack, i>` where `i` is `current` of `M`.

the new rules are

1. rule of constant



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

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

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

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

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),[])`

and store it the closure `(@x.(((+:7)(:1))3),[],6)` in `global_eval_env[6]` and increased `current` to 7.

for `MY 4`, the new rules will correctly access `MY` and returns 7

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{<3,[(4,[],7)],[],6>=>3}}\ \text{cst} \quad \dfrac{\dfrac{}{\texttt{<4,[],[],7>=>4}}\ \text{cst}}{\texttt{<(:1),[(4,[],7)],[],6>=>4}}\ \text{var}}{\texttt{<(:0),[(3,[(4,[],7)],6);((:1),[(4,[],7)],6)],[],1>=>7}}\ \text{op}}{\texttt{<@.(:0),[((:1),[(4,[],7)],6)],[(3,[(4,[],7)],6)],1>=>7}}\ \text{abs}}{\texttt{<@.(@.(:0)),[],[((:1),[(4,[],7)],6);(3,[(4,[],7)],6)],1>=>7}}\ \text{abs}}{\texttt{<(:7),[(4,[],7)],[((:1),[(4,[],7)],6);(3,[(4,[],7)],6)],6>=>7}}\ \text{var}}{\texttt{<((:7)(:1)),[(4,[],7)],[(3,[(4,[],7)],6)],6>=>7}}\ \text{app}}{\texttt{<(((:7)(:1))3),[(4,[],7)],[],6>=>7}}\ \text{app}}{\texttt{<@.(((:7)(:1))3),[],[(4,[],7)],6>=>7}}\ \text{abs}}{\texttt{<(:1),[],[(4,[],7)],7>=>7}}\ \text{var}}{\texttt{<((:1)4),[],[],7>=>7}}\ \text{app}$$

where `VAR (:0)` referes `(OP +)`.

3.3 不动点算子

3.3.1 Weak Head Normal Form

with CBV or CBN, we stop the evaluation if the AST top of the lambda term is an abstraction (weaker 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)x);
(@x.((@x.(x:1))(x:1))) |== (A -> A)
(@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) |== int
((@x.((@x.(x:1))(x:1)))1) => 1
>>> step = 7
```

3.3.2 严格与惰性不动点算子.

with those strategies, we can predefined the fixed-point combinators without any risk of the infinite loop of the preevalution:

```
let Y=@f.(@x.f(x x))(@x.f(x x)); /* lazy fixed-point combinator */
let Z=@f.(@x.f(@y.(x x)y))(@x.f(@y.(x x)y)); /* strict fixed-point combinator */
```

even the Church's numerial function with no lazy `IF` and `Z` can be predefined (see the definition in `library.txt`):

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

BUT CBV works only with our lazy `if-then-else-fi` and strict `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) |== int
((fact:1)10) => 3628800
>>> step = 332
```

because the evalution of `Y M` (`M` is any term) will go infinite loop in CBV, we can't predefine any recursive function with `Y` as fixed-point combinator, like :

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

>>>((: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:2913
```

`Y` and `Z`, even Church no-lazy `IF` work fine in CBN:

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

let fact = (Y (@f.@n. (if (= n 0) then 1 else (* n (f (- n 1))) fi)));
fact 10; /* lazy Y */
```

```
((fact:1)10) => 3628800
>>> step = 1129  /* fewer steps than Z */

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

/* Church numerial factorial */
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 TWO return the closure of @f.@x.f(f x). so FACT TWO (@x.+x 1) 0
   return 2 */

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` in CBN. in real world of functional programming language, we can't use this CBN.

3.3.3 交互递归

```
let EVEN = @f.@g.@n.if (= n 0) then 1 else g (- n 1) fi;
let ODD =  @f.@g.@n.if (= n 0) then 0 else f (- n 1) fi;

let ev = Z(@f.@n.EVEN f (Z(@g.@n.ODD (Z(@f.@n.EVEN f g n)) g n)) n);
let od = Z(@g.@n. ODD (Z(@f.@n.EVEN f g n)) g n);

ev 6;
((ev:2)6) |== int
((ev:2)6) => 1
>>> step = 394
```

see http://okmij.org/ftp/Computation/fixed-point-combinators.html in detail.

3.3.4 无限链表

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:12)(THREE:40))(FIBO:9))(@x.(((+:70)(x:1))1)))0) |== NULL
```

```
(((((GETN:14)(THREE:42))(FIBO:1))(@x.(((+:77)(x:1))1)))0) => 2
step = 1406
```

3.4 输出求值过程

Our sample program `./lambda` is compiled under UBUNTU X64. it can output the evaluation tree to LaTeX source file `evaltree.tex` (in prooftree format, `prooftree.sty` included). You can `pdflatex eval_a3l.tex` (A3 paper landscape) or `pdflatex eval_a4l.tex` (A4 paper landscape) to generate the pdf file (both in CBV and CBN mode).
Because the limitation of `prooftree.sty`, we only trace the first 50 steps of the evaluations. and the evaluation of large steps can cause `pdflatex` error.

3.5 内存泄露

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.

You can also use `valgrind` (http://valgrind.org) to check the memory leaks, like:

```
~$ valgrind --leak-check=full ./lambda
Call by NAME mode!!!
please input a lambda term with ";":
let Y=@f.(@x.f(x x))(@x.f(x x));

please input a lambda term with ";":
let facty = (Y(@f.@n. (if (= n 0) then 1 else (* n (f (- n 1))) fi)));

abs: []
type_env[] = [0(0,int), 1(1,(int -> int))]
((facty:1)3) |== int
((facty:1)3) |== int
((facty:1)3) => 6
>>> step = 170

please input a lambda term with ";":
CTRL+D
......

==3263== LEAK SUMMARY:
==3263==    definitely lost: 112 bytes in 6 blocks
==3263==    indirectly lost: 48 bytes in 3 blocks
==3263==      possibly lost: 0 bytes in 0 blocks
==3263==    still reachable: 22,380 bytes in 146 blocks
==3263==         suppressed: 0 bytes in 0 blocks
==3263== Reachable blocks (those to which a pointer was found) are not shown.
```

```
==3263== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==3263==
==3263== For counts of detected and suppressed errors, rerun with: -v
==3263== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
```

if we rerun it with `facty 7`, `valgrind` will summary the same results.

3.6 调用关系图

to visualize the gprof data, you can use `./gprof2dot` (a python script) (http://code.google.com/p/jrfonseca/wiki/Gprof2Dot) to transfer the `gprof` data to graphivz file(`.dot`), then visualize with dotty.

```
~$ ./lambda
......
CTRL+D

~$ gprof ./lambda |./gprof2dot.py -c gray|dot -Tpdf -o profile.pdf
```

see `profile.pdf` for detail.

you can also visualize the callgraph by `valgrind` like:

```
~$ valgrind --tool=callgrind --dump-before=main ./lambda
......
CTRL+D

~$ ./gprof2dot.py -f callgrind ./callgrind.out.4014 | dot -Tpdf -o callgrind.pdf
```

where `callgrind.out.4014` is the output of `valgrind`. see `callgrind.pdf` for detail.

4 TODO

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

please send your `closure.c` as attached file to `mailto:595180978@qq.com?subject=ID(06)` where the `ID` is your student id number.

–hfwang March 20, 2022

5 抽象机与编译

The abstract machine (see From Interpreter to Compiler and Virtual Machine: a Functional Derivation) is in fact is a serie of the instructions which can treat the evaluation of the last section. The compilation of the λ-term is in fact the **serializations** of the evaluation of CBV or CBN.

suppose our abstract machine (simplified SECD machine) is stack machine. its memory can hold the normal constant of integer, closure, instruction list and environment so called the **object**.

the machine state is triple `(pc, e, s)` where `pc` is program counter which represented as a list of instructions whose first element is the current instruction being executed, `e` is the register which always hold the current environment, and `s` the stack which stores the evaluation results.
the CBV evaluation will suggest the instructions of the abstract machine:

1. `NUM n`: push `n` on the stack
2. `GET n`: the register `e` must be the env, push `get_n_th(e, n)` to the stack.
3. `PUT c`: push `CLOS(c, e)` on the stack where `c` is the code of the function body and `e` is the current register.

4. APPLY: pop function argument and closure from stack, store current env and rest code as closure, and execute the code of the function.

5. RET: return caller env.

6. BRANCH: pop constant n, code of the true branch and code of the false branch, if n is no zero, then execute the concatenation of the code of the true branch and the rest code, else the false one.

7. the arithmetic op (ADD, SUB, MULT, DIV, EQ and LT): pop 2 constants, push the arithmetic result in stack.

formally, the code and the object (or value) are defined as (abstract syntax)

```
code ::= NUM of int
       | GET of int
       | PUT of code list
       | RET
       | APPLY
       | BRANCH
       | ADD | SUB | MULT | DIV | EQ | LT

object ::= CONST of int
         | ENV of object list  * int
         | CLOS of (code list) * (object list)
/* where int is the current for access let-binding's name
```

the data structures in C is (see code.h in detail)

```
typedef enum {NUM=1, GET=2, BRANCH=3, PUT=4, APPLY=5,
              RETURN=6, ADD=7, SUB=8, MULT=9, DIV=10,
              EQ=11, LT=12, TAIL=13} INSTRUCT;

typedef struct Instruction {
  INSTRUCT instruct;
  int value;
  struct Instruction *abs;
  struct Instruction *next;
} INSTRUCTION;

typedef enum {CONSTANT=1, CLOS=2, ENV=3} Object_kind;

typedef struct Object {
  Object_kind kind;
  int value;  /* for ENV, it's current for access let-binding's name */
  INSTRUCTION * instr_list;
  struct Object * env;
  struct Object * next;
} OBJECT;
```

the transitions table of the instructions:

Before			After		
pc	register	stack	pc	register	stack
(NUM n)::c	e	s	c	e	(CONST n)::s
(GET n)::c	e	s	c	e	get_n_th(e,n)::s
(PUT c')::c	e	s	c	e	CLOS(c',e)::s
RET::c	e	v::CLOS(c',e')::s	c'	e'	v::s
APPLY::c	e	v::CLOS(c',e')::s	c'	v::e'	CLOS(c,e)::s
BRANCH::c	e	(CONST n)::CLOS(t,l)::CLOS(f,l)::s	t++c	e	s
BRANCH::c	e	(CONST 0)::CLOS(t,l)::CLOS(f,l)::s	f++c	e	s
ADD::c	(CONST m)::(CONST n)::e	s	c	unchanged	(CONST n+m)::s
SUB::c	(CONST m)::(CONST n)::e	s	c	unchanged	(CONST n-m)::s
MULT::c	(CONST m)::(CONST n)::e	s	c	unchanged	(CONST n*m)::s
DIV::c	(CONST m)::(CONST n)::e	s	c	unchanged	(CONST n/m)::s
EQ::c	(CONST m)::(CONST n)::e	s	c	unchanged	(CONST n==m)::s

the SDD of the compilation

AST	semantic rules
T = CONST n	T.code = [NUM n]
T = VAR n	T.code = [GET n]
T = ABS(x, T1)	T.code = [PUT (T1.code ++ [RET])]
T = COND(T1, T2, T3)	T.code = (PUSH T3.code)::(PUSH T2.code)::T1.code::[BRANCH]
T = APP (T1, T2)	T.code = T1.code ++ T2.code ++ [APPLY]
T = Op +	T.code = [PUT [PUT [ADD; RET]; RET]]

like the typing and the evaluation, the global code execution environment is the extension of the environment, which hold the static part of the register. it is initializes as:

```
global_exec_env[] = [CLOS([PUT [ADD; RET]; RET], ([],0)), CLOS([PUT [SUB; RET]; RET], ([],0)),
                     CLOS([PUT [MULT; RET]; RET],([],0) ), CLOS([PUT [DIV; RET]; RET], ([],0)),
                     CLOS([PUT [EQ; RET]; RET], ([],0)), CLOS([PUT [LT; RET]; RET], ([],0))]
where ENV ([],0) is pair of (list of objects, current)
```

5.1 Example 1: + 1 2

the AST of + 1 2 is APP(APP(Op +, 1 2)) with current = 6, the compiled code is [GET 6; NUM 1; APPLY; NUM 2; APPLY].

step	PC	register	stack
0	[GET 6;NUM 1;APP;NUM 2;APP]	([],6)	[]
1	[NUM 1;APP;NUM 2;APP]	([],6)	[CLOS([PUT[ADD;RET];RET],([],0))]
2	[APP;NUM 2;APP]	([],6)	[CONST 1;CLOS([PUT[ADD;RET];RET],([],0))]
3	[PUT[ADD;RET];RET]	([CONST 1],0)	[CLOS([NUM 2;APP],([],6))]
4	[RET]	[CONST 1]	[CLOS([ADD;RET], ([CONST 1],0));CLOS([NUM 2;APP],([]),6)]
5	[NUM 2;APP]	([],6)	[CLOS([ADD;RET], ([CONST 1],0))]
6	[APP]	([],6)	[CONST 2;CLOS([ADD;RET], ([CONST 1],0))]
7	[ADD;RET]	([CONST 2;CONST 1],0)	[CLOS([], ([],0))]
8	[RET]	([CONST 2;CONST 1],0)	[CONST 3;CLOS([], ([],6))]
9	[]	([],6)	[CONST 3]

it's not efficent for application of 2 argument function, we must put the first application in closure. in fact for stack machine, we can simply do it by [NUM 1; NUM 2; ADD]. the abstract machine ZINC supports multiple argument's application.

5.2 Example 2. let MY = @x.+ x 3 (let-binding's name)

the code of MY is [PUT[GET:7;GET:1;APP;NUM 3;APP;RET]]. We pre-execute the code

step	PC	register	stack
0	[PUT[GET:7;GET:1;APP;NUM 3;APP]]	([],6)	[]
1	[]	([],6)	[CLOS([GET:7;GET:1;APP;NUM 3;APP;RET],ENV([],6))]

and put the result in global_exec_env[6]:

```
global_exec_env[] = [..., CLOS([PUT [LT; RET]; RET], ([],0)),
                     CLOS([GET:7;GET:1;APP;NUM 3;APP;RET],([],6))]
                     ----------------------------------------
                              global_exec_env[6]
```

and current increased to 7.

the compiled code of MY (@x.x) 4 is [GET:1;NUM 4;APP]

step	pc	register	stack
0	[GET:1;NUM 4;APP]	([],7)	[]
1	[NUM 4;APP]	([],7)	[CLOS([GET:7;GET:1;APP;NUM 3;APP;RET],([],6))]
2	[APP]	([],7)	[CONST 4;CLOS([GET:7;GET:1;APP;NUM 3;APP;RET],([],6))]
3	[GET:7;GET:1;APP;NUM 3;APP;RET]	([CONST 4],6)	[CLOS([],([],7))]
4	[GET:1;APP;NUM 3;APP;RET]	([CONST 4],6)	[CLOS([PUT[ADD;RET];RET],([],0));CLOS([],([],7))]
5	[APP;NUM 3;APP;RET]	([CONST 4],6)	[CONST 4;CLOS([PUT[ADD;RET];RET],([],0));CLOS([],([],7))]
6	[PUT[ADD;RET];RET]	([CONST 4],6)	[CLOS([NUM 3;APP;RET],([CONST 4],6));CLOS([],([],7))]
7	[RET]	([CONST 4],6)	[CLOS([ADD;RET],([CONST 4],0));CLOS([NUM 3;APP;RET],([CONST 4],6));CLOS([],([],7))])
8	[NUM 3;APP;RET]	([CONST 4],6)	[CLOS([ADD;RET],([CONST 4],0));CLOS([],([],7))]
9	[APP;RET]	([CONST 4],6)	[CONST 3;CLOS([ADD;RET],([CONST 4],0));CLOS([],([],7))]
10	[ADD;RET]	([CONST 3;CONST 4],0)	[CLOS([RET],([CONST 4],6));CLOS([],([],7))]
11	[RET]	([CONST 3;CONST 4],0)	[CONST 7;CLOS([RET],([CONST 4],6));CLOS([],([],7))]
12	[RET]	([CONST 4],6)	[CONST 7;CLOS([],([],7))]
13	[]	([],7)	[CONST 7]

in the step 3, the ENV is ([CONST 4],6), so the relative current is 6, and with the relative current, get_n_th_env(7) returns perfectly ADD:

```
OBJECT *get_n_th_env(OBJECT *env, int n)
{
  int i = 0;
  OBJECT *cursor = env -> env;
  int index = env -> value; /* relative current */
  while ( i != n -1 && cursor != NULL) {
    cursor = cursor -> next;
    i++;
  }
  if (i == n - 1 && cursor != NULL)
    return clone_single_object(cursor);

  if (index - (n - i) >= 0)
    return clone_single_object(global_exec_env[index - n + i]);

  printf("wrong access of closure env\n");
  exit (1);
}
```

5.3 尾调用

APPLY should store the current environmnent and the rest code as closure in stack for the coninuation. RET will jump to the prestored continuation. the code MY is [GET:7;GET:1;APP;NUM 3;APP;RET]. which means that the last operation of the body of the function MY is APPLY, after APPLY, the function will RET immediately to the caller. so this kind APPLY is not needed to save the continuation as closure, so called **tail call**.

We can change such tail call with a new instruction TAIL. it can signaficant saving stack space:

Before			After		
pc	register	stack	pc	register	stack
...	...	...	...	...	...
TAIL::c	e	v::CLOS(c',e')::s	c'	v::e'	s

5.3.1 Example 3. let MY = @x.+ x 3 with tail call

the code of MY is [PUT[GET:7;GET:1;APP;NUM 3;TAIL]]. We pre-execute the code

step	PC	register	stack
0	[PUT[GET:7;GET:1;APP;NUM 3;TAIL]]	([],6)	[]
1	[]	([],6)	[CLOS([GET:7;GET:1;APP;NUM 3;TAIL],([],6))]

and put the result in global_exec_env[6]:

```
global_exec_env[] = [..., CLOS([PUT [EQ; RET]; RET], []),
                      [CLOS([GET:7;GET:1;APP;NUM 3;TAIL],([],6))]
```

and `current` increased to `7`.

the compiled code of `MY (@x.x) 4` is `[GET:1;NUM 4;APP]`

step	pc	register	stack
0	[GET:1;NUM 4;APP]	([],7)	[]
1	[NUM 4;APP]	([],7)	[CLOS([GET:7;GET:1;APP;NUM 3;TAIL],([],6))]
2	[APP]	([],7)	[CONST 4;CLOS([GET:7;GET:1;APP;NUM 3;TAIL],([],6))]
3	[GET:7;GET:1;APP;NUM 3;TAIL]	([CONST 4],6)	[CLOS([],([],7))]
4	[GET:1;APP;NUM 3;TAIL]	([CONST 4],6)	[CLOS([PUT[ADD;RET];RET],([],0));CLOS([],([],7))]
5	[APP;NUM 3;TAIL]	([CONST 4],6)	[CONST 4;CLOS([PUT[ADD;RET];RET],([],0));CLOS([],([],7))]
6	[PUT[ADD;RET];RET]	([CONST 4],6)	[CLOS([NUM 3;TAIL],([CONST 4],6));CLOS([],([],7))]
7	[RET]	([CONST 4],6)	[CLOS([ADD;RET],([CONST 4],0));CLOS([NUM 3;TAIL],([CONST 4],6));CLOS([],([],7))]
8	[NUM 3;TAIL]	([CONST 4],6)	[CLOS([ADD;RET],([CONST 4],0));CLOS([],([],7))]
9	[TAIL]	([CONST 4],6)	[CONST 3;CLOS([ADD;RET],([CONST 4],0));CLOS([],([],7))]
10	[ADD;RET]	([CONST 3;CONST 4],0)	[CLOS([],([],7))]
11	[RET]	([CONST 3;CONST 4],0)	[CONST 7;CLOS([],([],7))]
12	[]	([],7)	[CONST 7]

(see `./lambda.tail` for detail)

5.4 TOTO

5.4.1 Compilation and Simulator

Please complete `INSTRUCTION *compile(AST * t)` (**only for CBV**) and the stepwise simulator `STATE *step_exe(STATE *state)` of `code.c`.

5.4.2 Optimization of tail call (BONUS)

Please design the new SDD so the tail call can be replace by `TAIL`, and implement your new `compile()`.

5.4.3 CBN (BONUS)

Please design your proper abstract machine and instructions for CBN, and implement compiler and simulator of CBN.

please send your `code.c` as attached file to `mailto:595180978@qq.com?subject=ID(07)` where the `ID` is your student id number.

–hfwang March 20, 2022