

Report for Lab03 – Garbage Collector

Stu:金泽文 No.PB15111604

使用语言：C++

环境 linux64 g++ 6.2.0 -std=c++11

运行方法：

编译方法：

make 或者

g++ Prj.cpp Term.cpp -o p -std=c++11 -w

运行方法：

./p 直接运行

./p 加上任意一个参数，导出处理之前与处理之后的语法树结构，可发现命名变化。

./p 加上任意两个参数，导出处理之前与处理之后的语法树结构，可发现命名变化。输出行号。（调试用）

测试样例见文件 examples.txt

下面是关于 gc 算法的说明。

Garbage Collector 的实现

具体思路：

由于我的解释器整体是在语法树上进一步生成的，所以思路跟别人的“活动记录”实现方式有比较大的不同。

典型的 gc 算法，如标记-清除，引用计数，复制算法，等统统都没有用。由于我们要实现的语言功能很单一，没有指针的概念，所以只需要在解释的过程确定之后不会再用即可清除，释放空间。

我是针对每一块语法结构而设计的算法。

①针对**函数调用**，②针对 Assign，Read，Print，If，While 等 **Command**。

为了释放空间，我设计了两个函数，deleteMem 和 deleteTree，后者被前者调用，前者通用于以上所有的情况释放节点所用。

①针对**函数调用**：由于我的函数是在每个的 apply 和 call 的 sons 链表中 push_back 一个子树来复制函数体并且调用，所以我释放函数是在函数调用完毕之后释放 sons.back()。

②针对 Command：由于这些 command 在一次解释之后就不再有作用，所以直接释放，通过 deleteMem。

需要注意的是，由于 While 的存在，所以在释放的过程中，要考虑之后是否要复用，所以，call 之后利用 underWhile 函数判断是否要复用，如果不服用，则把 call 节点也释放掉；command 解释之后，也利用 underWhile 函数判断是否需要复用，如果需要复用，则不能释放。

另外，对于 Block 中的变量作用域的问题，由于 block 只能在 if，while，call，apply 中出现，而这四个我已经考虑了，所以不需要进一步考虑。

最后，我写了几个测试样例，并通过设置 maxMem (内存峰值), memCount (当前内存), totalMem (总共分配过的内存) 三个变量来表现我的 gc 的效果。

针对测试样例，我是在 BashOnWindows 上跑的，内存占用是在 windows 的任务管理器上观察的。

样例 1：

```
1 Begin
2   Var x End
3   Function fact Paras i
4     Begin
5       If Lt i 2
6         Begin
7           Return 1
8         End
9       Else
10      Begin
11        Return Apply fact Argus Minus i 1 End
12      End
13    End
14    Read x
15    Print Apply fact Argus x End
16    Read x
17    Print Apply fact Argus x End
18    Read x
19    Print Apply fact Argus x End
20    Read x
21    Print Apply fact Argus x End
22    Read x
23    Print Apply fact Argus x End
24    Read x
25    Print Apply fact Argus x End
26 End
27
```

开始：等待输入：目前内存 0.2MB

| | | | |
|---|----|--------|----------|
| evin@KZ /mnt/f/csapp/大二下/FoPL/project/F0PL_project_2017 | | | |
| % ./p | | | |
| NVIDIA Streamer Service | 0% | 0.3 MB | 0 MB/秒 |
| NVIDIA Streamer User Agent | 0% | 1.9 MB | 0.1 MB/秒 |
| NVIDIA User Experience Driver Component | 0% | 0.3 MB | 0 MB/秒 |
| > O2 Flash Memory Service (32 位) | 0% | 0.1 MB | 0 MB/秒 |
| P | 0% | 0.2 MB | 0 MB/秒 |
| plugin host exe (32 位) | 0% | 0.2 MB | 0 MB/秒 |

输入 8000，输出 1，等待输入：可以看到，目前内存 14.9MB

| | | | |
|---|-------|---------|----------|
| % ./p | | | |
| 8000 | | | |
| 1 | | | |
| NVIDIA Streamer Service | 0% | 0.3 MB | 0 MB/秒 |
| NVIDIA Streamer User Agent | 0% | 1.9 MB | 0.1 MB/秒 |
| NVIDIA User Experience Driver Component | 0% | 0.3 MB | 0 MB/秒 |
| > O2 Flash Memory Service (32 位) | 0% | 0.1 MB | 0 MB/秒 |
| P | 29.1% | 14.9 MB | 0 MB/秒 |

再输入 5000，输出 1，等待输入：可以看到，内存还是 14.9MB，没有增加。

| | | | |
|---|-------|---------|----------|
| evin@KZ /mnt/f/csapp/大二下/FoPL/project/F0PL_project_2017 | | | |
| % ./p | | | |
| 8000 | | | |
| 1 | | | |
| 5000 | | | |
| 1 | | | |
| NVIDIA Streamer Service | 0% | 0.3 MB | 0 MB/秒 |
| NVIDIA Streamer User Agent | 0% | 1.9 MB | 0.1 MB/秒 |
| NVIDIA User Experience Driver Component | 0% | 0.2 MB | 0 MB/秒 |
| > O2 Flash Memory Service (32 位) | 0% | 0.1 MB | 0 MB/秒 |
| P | 31.3% | 14.9 MB | 0 MB/秒 |

再输入 8000，输出 1，等待输入：同上，没有增加。

| | | |
|---|-------|---------|
| % ./p | 49% | 55% |
| 8000 | CPU | 内存 |
| 1 | | |
| 5000 | | |
| 1 | | |
| 8000 | | |
| 1 | | |
| NVIDIA Streamer Service | 0% | 0.3 MB |
| NVIDIA Streamer User Agent | 0% | 1.9 MB |
| NVIDIA User Experience Driver Component | 0% | 0.2 MB |
| > O2 Flash Memory Service (32 位) | 0% | 0.1 MB |
| P | 37.9% | 14.9 MB |

最终：

| | | | |
|----------------------------|--|--|--|
| % ./p | | | |
| 8000 | | | |
| 1 | | | |
| 5000 | | | |
| 1 | | | |
| 8000 | | | |
| 1 | | | |
| 任务管理器 | | | |
| 文件(E) 选项(O) 查看(V) | | | |
| 进程 性能 应用历史记录 启动 用户 详细信息 服务 | | | |
| 名称 | | | |
| maxMem = 5 M 1008 K 288 B | | | |
| totalMem = 19 M 84 K 624 B | | | |
| memCount = 1 K 264 B | | | |

样例 2：

```
1
2 Begin
3   Var x End
4   Var y End
5   Assign y 2
6   Function fact Paras
7   Begin
8     Function t Paras Begin
9       Return 1
10    End
11    Return t
12  End
13  Read y
14  While Lt 1 y
15  Begin
16    Assign x Apply fact Argus End
17    Assign y Minus y 1
18  End
19  Print Apply x Argus End
20  Read y
21  While Lt 1 y
22  Begin
23    Assign x Apply fact Argus End
24    Assign y Minus y 1
25  End
26  Print Apply x Argus End
27
28  Read y
29
30  While Lt 1 y
31  Begin
32    Assign x Apply fact Argus End
33    Assign y Minus y 1
34  End
35  Print Apply x Argus End
36
37 End
38
```

输入 8000,8000,80000.

效果如右图。

```
evine@K2: /mnt/1/csapp/大二下/POPL/pro
% ./p
8000
1
8000
1
80000
1
maxMem   = 4      K  720  B
totalMem = 66     M  668  K  472  B
memCount = 1008  B
evine@K2: /mnt/1/csapp/大二下/POPL/pro
```


下面是前两次实验过程的记录，供助教参考。

前两次实验过程的部分文档记录

对于 prj1，我实现的方式与细节：

1.函数：

return 0：

考虑到默认的 return 0 在语法树处理之后自动添加 return 0 利用的是 setFunction 和 addReturn 函数。

return 的实现：

利用 searchFun 函数找到调用的地方，即 call 和 apply 的节点。并将值赋给这给节点的 number。

declare：

无视 declare

call，apply：

找到调用的函数，根据 searchName

对于 Argus，参数值通过 expr 得到，并赋值。

对于 function block 部分，采用 deepcopy 的模式。

并且由于要 copy，所以参数列表的参数名必须与声明函数时一致。所以要换名字。

总体用 callFunction 函数实现

2.static scope:

将所有的变量名按照所声明的指针值重命名，

先解决使用变量时的名字在。

call apply 或者 expr 时

的变量名根据 static_searchName 和 changeUsedName 函数

再解决 def 变量。

根据 changeDefName 函数。

遍历一律采用先序遍历。

3.语法树结构：

Begin End -> k=Block,s=Declaration

Var x End -> k=Command,s=Declaration

-> k=Name,s=Declaration,number,name

Assign x 1 -> k=Command,s=Assign

-> k=Name,s=Declaration,number,name

-> k=Expr,s=Number,number,name=""

Assign x y -> k=Command,s=Assign

-> k=Name,s=Declaration,number,name

-> k=Expr,s=VarName,number,name="y"

Function g Paras i-> k=Function,s=Declaration

-> k=Name,s=Declaration,number,name

-> k=Name,s=Declaration,number,name

-> k=Block,s=Declaration

-> ...

Read x -> k=Command,s=Read

-> k=Name,s=Declaration

Call h Argus y End -> k=Command,s=Call,

-> k=Name,s=Declaration,name

-> k=Expr,s=VarName,number,name

Prj2 高阶：

things to be solved

1.return function

--只有 apply 的时候才会用到 return

应该在 apply 返回时设置一个 flag 变量，

标志是 int 还是 function，以便确定后面的内容。

2.bind the var to the function

--为方便起见，我根据 assign 的结果，

比如函数还是 int，

选择去掉 Var x End，并弄一个 Function Dec，

还是保留 Var x End。

每次 assign 都选择一次

int 依旧保留 var，

赋值的是 function 则再次重新 function declaration

改完之后直接弄到 father 的 sons 中的对应次序。

3.Print--maybe should be considered?

4.Return->int or function

5.Assign function to var

--can be replaced by the Function Declaration

[5-20-23:07]保存了 prj1 的代码，以备后患。

[5-21-00:09]发现，只能通过 expr 判断出 int 还是 fun，

因为，print、apply、return 都只能通过后面表达式的返回值才能够判断。

[5-21-15:35]return，assign 的时候，

将返回的函数的指针，赋值到对应节点中。

return 返回函数只在 apply 的时候用到，

就赋值到 apply 这个节点的 pointToFun。

assign

[5-21-15:46]决定不用 pointToFun，而是直接用 number。

采用强制类型转换。

[5-21-16:13]再返回函数的时候，应该返回到 callfunction 的地方。

[5-21-16:15]return 时，无论 VarName，还是 Apply，都需要构造。

而且要注意换名字。

[5-21-16:18]当 expr 中 VarName 并且 vartype 为 Fun 时，

有 Assign f h 的情况，

有 Return h 的情况。

都需要构造。不如直接在 expr-VarName-Fun 时直接 create。

另外，return apply 返回函数时也是 return，

所以 expr-Apply-Fun 时也直接 creat

[5-21-16:33]检查一下赋值，call，apply 的时候参数等等的 intfun 有没有一起赋值

[5-21-17:03]想起来 Print 的 VarName 并解决。

[5-22-14:54]发现根源的需要 createFun 的地方，只有 return f 的时候。以及 assign f 的时候

[5-22-15:24]有两个地方要改名字。

一个是 assign a f 的时候的函数名，

只有 assign 的时候才知道最后的函数名。

还有一个是 apply 的时候的参数名

[5-22-15:53]发现，createFun 的时候，参数 fun 是在 apply 或者

[5-22-16:27]所以最后考虑的两种情况

1.assign x f，

这时根据

assign->sons.back()->subtype==VarName

以及

```
assign->sons.back()->vartype==Fun
```

2.return t , 包括 assign x Apply f Argus End

一个特例：

```
apply apply x Argus End Argus 1 End
```

但是语法通不过，所以不考虑。

这时根据

Apply->father->sons.front()得到赋予的函数名，即 x

[5-22-16:58]完成了上述情况 1 的处理，还差 2

[5-22-17:12]弄一下参数是函数的情况

```
Function f Paras g h
```

```
Begin
```

```
Function t Paras i
```

```
Begin
```

```
Call g Argus End
```

```
return Apply t Argus
```

```
End
```

```
Return t
```

```
End
```

```
Assign x Apply
```

//想找一个使用函数作为参数，而不在函数内部使用 apply 或者 call 该函数参数的情况

但是没想到。

[5-22-17:25]再弄情况 2

[5-22-21:26]跑了一下测试 1，调试之后通过了

[5-22-23:41]又想到一个例子。关于判断是否参数

[5-23-00:48]情况 1 跑完。

[5-23-01:04]测试完 argu 替换

[5-23-01:31]想到一个参数为函数，将其赋值给外部变量的例子。发现 bug

[5-23-01:46]解决，正在弄 return t 的问题

[5-23-02:33]写完 prjt2