对``小语言学家"的简要介绍, 以及它的渐进式垃圾回收器的简单而有趣的实现

谢宇恒

2014年5月12日

摘要

在下面的文字中,我介绍我所实现的一个渐进式垃圾回收器,

我改进了三染色算法以增强其渐进性.

这个垃圾回收器是我用我所设计的一个栈处理语言来写的.

这个语言的名字叫``小语言学家" (即``xiaolinguist")

它是我受Forth和Scheme还有Joy的启发而设计的.

关键词: 程序语言, 函数式编程, 垃圾回收器.

Abstract

In the following text, I present a incremental garbage-collector.

I use tri-color algorithm to implement this garbage-collector,

and I improve this algorithm to make it more incremental.

This garbage-collector is implemented in a language designed by me,

I call my language ``xiaolinguist",

``xiao" is a Chinese character denotes ``little".

My design philosophy is influenced by Forth Scheme and Joy.

Key words: programming language, functional programming, garbage collector.

目录

动机

这里所说的``动机",指的是我去设计并实现``小语言学家"这个程序语言的动机, 而不是我给这个语言实现一个垃圾回收器的动机. 后者在我看来是自然的.

我所感兴趣的是程序语言背后的计算模型,即数学理论方面. 我了解了表处理语言Lisp(list processing language), 还了解了列表处理与λ-calculus这个计算模型之间的有趣关系[Henk Barendregt, 1997, 2000, 1980]. 我想要推广λ-calculus这个计算模型本身, 类比之下,我需要实现一个有向图处理语言(directed graph processing). 这个语言将适合于用来 把一个人关于计算模型的任何有趣的数学念头付诸实践.

之初,我用Scheme这个Lisp方言来实现有向图处理语言. 我赞赏设计上的极简主义哲学所赋予Scheme的简洁与优雅, 并且我喜爱Scheme灵活而强大的抽象能力. 但是为了达到高度的抽象性Scheme放弃了对底层的灵活控制, 我发现这对于我实现我所希望的有向图处理语言而言是个障碍.

这在于, ``把一个有向图保存在内存中,然后用函数去改写这个有向图." 并不是Scheme所遵循的函数式编程范式所擅长的工作, 在函数式编程的术语中, 这种函数叫做带有副作用(side-effect)的函数, 对它们的使用是出离函数式编程范式的, 因为这使得 **程序中的函数** 不能被当作一个 **数学意义上的函数** 来被理解与分析. 也就是说副作用的使用破坏了``函数语义".

Scheme所擅长的是(以链表处理为例)[Daniel P. Friedman 与 Matthias Felleisen, 1996]: ``一个函数接收一个链表,然后它解构这个链表, 然后利用解构来的零件构造一个新的链表. 这个新的链表将被当作返回值, 而所输入的链表将被抛弃." 这样的函数显然可以被理解为数学意义上的函数. 但是当链表被作为参数在函数之间传递的时候, 所传递的当然是对链表的引用,而不是链表的值(那将需要复制整个链表). 因为一个链表的引用可能被作为参数传递给多个函数, 所以如果其中一个函数用副作用改变了被引用的链表在内存中的存储状态, 那么另一个同样引用这个链表的函数可能就会出错了. 所以上面那句话应该被重述为: ``一个函数接收一个链表,然后它遍历这个链表, 然后利用遍历时所得到的信息构造一个新的链表. 这个新的链表将被当作返回值." 只要没有副作用,那么用哪种方式来理解这个函数的作用过程都是可以的. 但是必须强调 **没有副作用**.

Scheme所擅长的工作能够完成的任务非常多. 比如在Scheme中,我可以在100行代码以内写一个Scheme的解释器; 我也可以用2000行左右的代码写一个高度优化的Scheme编译器(的核心部分). 这在于这些任务所要处理的数据是语法解析树(在Lisp的术语中叫做S-exp,代表Symbol-expression), 它们都是具有良好的 **归纳定义** 的, 这样我就很容易用递归函数来处理这些数据, 因此上面那句话应该被重述为: ``一个函数接收一个链表,然后它 **递归地** 遍历这个链表, 然后利用遍历时所得到的信息 **递归地** 构造一个新的链表. 这个新的链表将被当作返回值."

然而,与 定义一个程序语言之语法的 语法解析树不同, 我想要处理的有向图的集合没有良好的归纳定义. 这就遇到了我所认为的Scheme的局限性. 我在Scheme中实现了一个有向图的数据结构, 并实现了很多处理有向图的基本函数, 但是我并不满意那些代码, 只因为那些代码看起来过分复杂与笨拙, 而其中很多表达上的复杂性是我所使用的语言的局限性所导致的, 而不是内蕴于我希望解决的问题中的. 我知道这些局限性是什么, 所以我想设计一个我自己的Scheme实现(Scheme implementation), 看看我能不能消除这些局限性. 所以我需要写一个新的Scheme编译器,并且在这个编译器中实现一些实验性的想法. 为了写一个编译器,我需要一个中间语言, 于是我找到了Forth这个具有灵活的底层控制能力的栈处理语言.

我很快实现了一个新的Forth,它与古典的Forth有很大的区别. 在使用这个新的语言的过程中我发现,我不再需要回去写Scheme的编译器了, 因为我所实现的新语言 不仅有对底层的控制能力, 同时它也(可以)有像Scheme一样强的抽象能力. 于是我把它命名为``小语言学家",

- 因为 它的核心部分本身没有语法, 而我,作为用户,很容易通过在其中定义一个一个的小编译器,来实现我所希望的任何语法;
- 又因为 与Scheme一样它的设计也将遵循极简主义哲学;

- 又因为 一个人很容易就能透彻地理解它作为一个机器的行为方式, 就像一个玩具一样,我甚至能把它的原理讲授给一个初中生.

Lisp的List

Lisp被称作链表处理语言, 因为它的主要数据结构是(被动态内存管理机制管理起来的)链表.

Lisp有很多方言,它们的性状集合和设计哲学可能都互不相同, 但是总是具有上面这个共同点.

Lisp把它对单向链表的实现方式暴露给了用户 [R5RS, 1998], 也就是说,用户可以用`cons`函数来构造pair, 并用`car`与`cdr`函数来分别取出pair中保存的第一个元素与第二个元素.[1]

链表(尤其是在嵌套的时候)常被称作S-exp(简写为sexp). 因为是由pair构造而成的, 所以sexp常常被视为``有向二叉树", 所以它的渐进式gc算法被称为``有向二叉树的三染色算法".

但是用 **副作用** 可以随意地在图中形成 **圈**, 这样就可以形成非单连通的有向图, 所以对gc算法的这种命名其实是错误的. 保留``sexp"这个术语的同时, 我将直接称gc算法为``三染色算法".

栈处理语言的函数式编程

与列表处理语言一样,栈处理语言也有一个语言家族,其根源在Forth [Charles H. Moore, 1970].

首先把函数式编程范式引入Forth语言族的是Joy [von Thun. Manfred, 2001]. 其作者让我们看清了[2]所有的计算过程都可以被表达在一个结合代数之内.[3] `1 2 swap +` 在古典的Forth中被理解为:把1入栈;把2入栈;交换栈中的头两个元素; 取出栈中的头两个元素,把它们的和入栈. 而在Joy中一切都被理解为函数,它们唯一的参数是栈唯一的返回值也是栈, 这样,1就是一个函数,它接受一个栈,返回一个栈,这个返回栈的栈顶多了自然数`1', 上面的表达式也就是四个函数的复合了.

在Lisp中,(主要的)[4]抽象能力由λ-calculus中的λ-abstraction来提供. 而在Joy中,(主要的)[5]抽象能力由combinator[6]提供. 并且Joy作者认为,函数复合语义 优于 函数作用语义,而 组合逻辑 优于 λ-calculus, 其论据是一元函数之间的复合满足结合律.

---

[1] ``cons"指 ``construct", ``car"指 ``Contents of the Address part of Register number", ``cdr"指 ``Contents of the Decrement part of Register number", 这里的``register"指``memory location", 这些有趣的术语与50年代末用来实现Lisp的硬件有关.

[2] 这是如此显然. 但是当一个人看不出一个显然的事实的价值与意义的时候, 这个人就还没能完全看清这个显然的事实.

[3] 可以说Forth这种栈处理语言 提供了一个世界上最有趣的结合代数, 比如: 任取一个 有限阶置换群, 都存在 这个有趣的结合代数的子代数 与 所取的有限阶置换群 同构.

[4] 另外一种抽象能力由macro提供.

[5] 不同的Forth方言提供抽象能力的方式不同, 但是最基本的 被公有的 方式是用函数的复合来定义一个新的函数. 不同的Forth方言会提供不同的形成高阶函数的方式.

[6] 在λ-calculus中,一个combinator是一个不含自由变元的λ-term. 因为不含自由变元, 所以这个λ-term可以被看作是 单纯地在描述函数与函数之间的一种组合方式, 这样也许就是combinator这个名字的来历(但是我没有费心去考证). 比如Y-combinator可以用来返回匿名的递归函数.

我想我可以用下面的话来总结两个语言族的特点: Lisp的语法(S-exp)是就 **函数作用** 而优化的, 想要在Lisp中表达 函数的复合 就要费一些笔墨(费一些字符); 而Forth的语法是就 **函数复合** 而优化的, 想要在Forth中表达 函数的作用 就要费一些笔墨.[7]

这种启示非常有趣,当我们改变我们的表达方式的时候,我们对我们想要表达的东西的认识也跟着改变了.[8]

比如: 对二元运算的结合性的证明,可以被转化为对一种特殊的交换性的证明:

```
infix:
   (p+q)+r == p+(q+r)
postfix:
   pq+r+ == pqr++
   +r+ == r++
```

那么,如果有某个星球的外星人总是使用后缀表达式, 那么他们也许就会把我们的``加法结合律"命名为``加号前面的加号和数字的交换律". 不知道这个星球的外形人的数学会比我们进步还是落后, 但是它们设计程序语言时一定比我们省心多了(如果他们也有`心'的话).

``加法交换律":

```
infix:
   a+b == b+a
postfix:
   ab+ == ba+
```

就被外星人称作``加号前数字的交换律".

``乘法对加法的分配律"或者说``乘以一个数' 作为整数集上的变换 是整数加法群的同构变换":

```
infix:
   (a+b)*c == (a*c)+(b*c)
postfix:
   ab+c* == ac*bc*+
   ab+(c*) == a(c*)b(c*)+
```

就被外星人称作``乘一个数' 与加号的双倍交换律".

我们用基本的运算律来给数学表达式做恒等变形, 每当看到一个数学表达式的时候, 我们的想象力就会帮我们 利用我们熟悉的运算律 来改变这个表达式的形状. 然而,用后缀表达式时,所有的运算律都被表达成了 **特殊的交换律**, 这个特点非常有趣.

关于垃圾回收器

对程序语言的设计者和实现者来说, 垃圾回收器(Garbage-collector,简称gc)总是一个有趣的问题. 其有趣就在于有,有非常多的算法可以选择,而不同的算法又可以组合使用, 这样就产生了性状迥异的很多 gc .

- 标记并清扫(mark-and-sweep):

[7]我也可以说Lisp和Forth根本就没有语法, 在编译或解释它们的代码的时候都不需要语法解析. 因为S-exp就是语法解析树, 而Forth中使用的 后缀表达式 可以被看成是反过来的语法解析树(如果我把前缀表达式定义为``正向"的话).

[8]有人认为中缀表达式(infix)是最自然的表达方式, 但是其实 对于被当作二元运算的二元函数来说, 只有当这种二元运算满足结合律的时候使用infix才是令人满意的, 比如: + * max min gcd `函数的复合' `字符串的并联' 等等. 而当二元运算不满足结合律的时候, 就算是在传统的数学表达式中 也是不使用中缀表达式的, 比如: `方幂' `根号' `取对数' 等等. 人们正是用这种非对称的表达式来强调 某些二元运算的 非结合性 与 非对称性 的.

这是最早的gc算法,由John McCarthy在创造Lisp的时候首次提出.[John McCarthy, 1960] 由于pair的大小是固定的,所以所有被分配的pair构成一个数组(而不是一般的堆). 每个pair上留有一个位置用来做标记.

所有自由而能够被分配的pair被串联成一个自由链表(free-list), 每次当自由链表被分配完了之后, 以所有全局变量和局部变量为根节点, 遍历所有能够被引用到的pair并标记它们(由于上一节所指出的原因, 我不能说所遍历的是一个有向二叉树或森林); 然后扫描所有pair所形成的数组,把其中没有被标记的pair重新串联到自由链表中, 并把其中被标记的pair上面的标记擦掉,以便下一次gc的工作.

因为有一个明显的清扫过程,所以这可能就是``垃圾回收器"这个名字的由来.[John McCarthy, 1960, 第四节注脚]

- 复制并压缩(copy-and-compress):

这是对标记并清扫算法的改进, 在动态管理pair这种定长度的内存块的分配之外, 同时可以以一致的方式来动态管理字符串和向量这种变长度的内存块的分配. 使用两个堆, 在标记的过程中,把所遍历到的元素从一个堆复制到另一个堆, 并更改所有引用到这个元素的 变量或其他链表 的引用点(如果不使用特殊的技巧的话, 这可能很耗时间). 给每一次做标记的时间加上了一个常量, 但是把分配一个pair平均时间降低到了O(1/(M+1)), 其中M是堆的大小,所以 Andrew W. Appel 得出有趣的结论[Andrew W. Appel, 1987]: ``如果所能使用的内存空间足够的大,那么垃圾回收器的速度可以比栈快". 但是这只是一个有趣的理论上的结果, 在实践中其真实性是有争议的.[Robert Hieb 等人, 1990, 第二节;第六段]

- 引用计数(reference counting):

不能简单地处理循环引用, 并且每次改变所记录的引用次数的过程可能很耗费时间. Unix文化圈里的人经常使用这中方式来实现垃圾回收器,比如Perl和Emacs-Lisp.

- 三染色(tri-color)算法:

由Dijkstra首先提出 [Edsgar W. Dijkstra 等人, 1976],使gc获得了渐进性, ``渐进性"的意思就是需要垃圾回收的时候不必等很长时间. 在后面的文字中,我会描述它,并且说明我对它的改进.

小语言学家的简介

作为栈处理语言,``小语言学家"有四个基本的栈[9]:

- ArgumemtStack 用于参数传递
- ReturnStack 用于函数的调用与返回
- GreyPairStack 用于gc的三染色算法
- LambdaStack 用于形成 函数作用 语义

[9]Lisp被称作链表处理语言, 那是因为它以链表为基本数据结构, 在其中,表达函数的是链表(S-exp),表达数据的也是链表(S-exp). 又例如 当APL被称作数组处理语言时, 那是因为它以数组为基本数据结构, 它拥有很多能力强大的 处理高维数组的函数. 然而 当 小语言学家 和 Forth 被称作栈处理语言时, 所指 却很不一样,它们并不以栈为基本数据结构, 这里的 所指 是: 它们把传递参数所用的栈, 还有函数返回所用的栈 都暴露给了用户, 并且提供了一些有趣的函数来处理这两个栈. 所以我可以说, ``某某某处理语言"或者``面向某某某的语言"这两个术语是被滥用了的.

用户通过一个解释器来与``小语言学家"交流. 解释器的交互界面在Lisp的术语中叫做REPL, 即 read-evaluate-print-loop, 阅读器读表达式,然后求值它,然后把返回值打印出来,然后循环. 但是因为作为类Forth语言, ``小语言学家"会把返回值返回到ArgumemtStack中, 需要另外调用各种函数来打印ArgumemtStack中的值; 并且这里也没有需要被求值的表达式, 所以REPL变成了 read-execute-(maybe-print)-loop.

解释器会一个词一个词的来读用户的输入, 词语词之间用空格或空行隔开. 可以以Joy所引入的函数式编程语义来理解每个词; 也可以用古典的Forth的方式来理解每个词, 在在面的例子中,我将使用后者.

词分为三类: 数字是一类; 常量与变量名被认为是 **名词**, 我约定它们的首字母大写(就像德语中的名词一样); 函数(或者称子程)名被认为 **动词**, 我约定它们的首字母小写.

有一个字典用来保存所有名词和动词, 名词和动词的定义都在字典中查找.

解释器遇到数字时会直接把这个数字入栈; 常量名会把所对应的常量入栈; 变量名会把所对应的变量的地址入栈, 用fetch和save这两个函数可以对变量进行操作; 解释器遇到函数名就调用这个函数, 函数的参数来自于ArgumemtStack, 返回值返回到ArgumemtStack; 特殊的函数是负责输入与输出的各种reader和writer, 一个reader的参数常常是它后面的某类字符串, 也就是说, 其参数可以不光来自与它前面的词(这些词所入栈的值), 还可以来自于它后面的词(或字符串).

下面的例子定义了阶乘函数:

```
: factorial (* n -- n! *)
    duplicate one? if
      Exit
    then
    duplicate sub1 factorial *
    Exit
; defineRecursiveFunction
```

其中`(* n -- n! *)`是注释, 用来说明函数对栈的影响,这也可以被看成是函数的类型, 例如 上面的注释表明`factorial`这个函数从栈中取一个参数n, 而把n!返回到栈中; 例如 `duplicate`的函数类型是 `(* n -- n, n *)`, 其意义是显然的; `:'是一个reader, 它把`;'之前的词都读到一个缓冲区中, 并把 冲区的地址 和 所读到的词的个数返回到栈中; `defineRecursiveFunction`把缓冲区中的词的列表编译到字典中, 我把这类扩展字典的函数称为 **字典编撰者**. 其他的 字典编撰者 有: `defineFunction defineVar defineConst defineArray defineConstString`, 其中只有最基本`defineFunction`是在汇编中定义, 其他的 字典编撰者 都是被它来做为函数而定义的,比如:

```
: defineRecursiveFunction (* wordList[address, the number of words] -- *)
    tailAndHeadOfWordList
    createWordHeaderForFunction
    setTheSizeOfFunctionBody
    addNewWordToDictionary
    FunctionBodyExplainer appendNumberToHere
    appendWordDescriptionToHere
    Exit
; defineFunction
```

用汇编代码写的, 最基本的`defineFunction`并不能使用`if ... then ...`这对语法关键词, 而只有`branch`和`false?branch`这种条件转条和无条件转条, 下面的40多行代码[10]重新定义它以增加这两个基本的语法关键词:

```
: appendWordDescriptionToHereWith:if&then
```

[10]其中loop是由尾递归函数形成的, 因为有为递归优化(tail-call-optimization), 所以尾递归函数的空间复杂度是常数. 频繁使用递归函数的语言, 如果没有尾递归优化就很容易让ReturnStack溢出. 其实所谓``为递归优化"根本不是编译器优化, 也不涉及对代码的静态分析, 只需要判断是否已经到达函数体的尾部, 并在到达时, 不要让对 尾部函数 的调用还返回到这个函数体内就行了. 我很难理解为什么有的语言(比如Python)没有尾递归优化. (Python作者的解释是:`好程序员'不使用递归函数.)

```
  (* wordList[address, the number of words] -- *)
  duplicate zero? false?branch 3
    twoDrop Exit
  tailAndHeadOfWordList
 twoDuplicate String,KeyWord,if equalString? false?branch 12
    twoDrop
    literal false?branch appendNumberToHere
    Here fetch xx|swap|x
    duplicate appendNumberToHere
    appendWordDescriptionToHereWith:if&then Exit
 twoDuplicate String,KeyWord,then equalString? false?branch 8
    twoDrop
    x|swap|xx over swap subSave
    appendWordDescriptionToHereWith:if&then Exit
  twoDuplicate stringDenoteNumber? false?branch 6
    number drop appendNumberToHere
    appendWordDescriptionToHereWith:if&then Exit
    find wordLinkToWordExplainer appendNumberToHere
    appendWordDescriptionToHereWith:if&then Exit
; defineRecursiveFunction

: defineRecursiveFunctionWith:if&then
  (* wordList[address, the number of words] -- *)
  tailAndHeadOfWordList
  createWordHeaderForFunction
  setTheSizeOfFunctionBody
  addNewWordToDictionary
  FunctionBodyExplainer appendNumberToHere
  appendWordDescriptionToHereWith:if&then
  Exit
; defineFunction

: defineFunction
  (* wordList[address, the number of words] -- *)
  tailAndHeadOfWordList
  createWordHeaderForFunction
  setTheSizeOfFunctionBody
  xx|swap|x
  FunctionBodyExplainer appendNumberToHere
  appendWordDescriptionToHere
  appendWordDescriptionToHereWith:if&then
  Exit
; defineFunction
```

每个 字典编撰者 都是一个小编译器, 上面的例子表明了 很容易在``小语言学家"中扩展它的编译器本身.[11]

Forth的作者(Charles H. Moore)发表过类似[12]下面这样的有趣见解: ``别用Forth写程序!而用Forth来定义一些词. 当你想要实现某个有趣的想法的时候, 你就写一百个词来讨论这个想法, 然后用这一百个词写一个简短的定义来实现你的想法或者解决你的问题. 想要找出这一百个词并非易事,但是它们存在,它们总是存在的!"

[11]`:'和`;'这对组合 会把其间的词的序列读到 一个固定的缓冲区当中, 这样 字典编撰者们 的编译过程就不灵活. 当实现gc之后, 我还添加了`::'和`;;', 每次使用它们,都将把一个词的序列读到一个新的被gc动态分配的链表当中, 之后我就能很容易地定义 能够处理更丰富的语法的 字典编撰者. 这种 实现新语法的简单性 正是小语言学家的特点.

[12]我并没有单纯地引用或翻译,所以只能说是``类似".

Forth社区中有一个术语叫``factoring"[13], 意思是去不断的调成你对一个函数的定义方式, 分解已有的定义,重构这些定义,以获得更恰当的定义. ``factoring"正是寻找那``一百个词"的过程.

在 小语言学家 与 Forth 中, 你是通过 ``**作定义,以明显地命名**" 这种方式 来形成对一个函数的各种不同的分解的.

当你需要定义一个函数来完成一个任务或解决一个问题, 在定义这个函数之前你就知道, 这个函数一定将是很多函数复合而成的, 所以你的任务就是去寻找一种将其他的函数复合起来的方式, 以形成一个能完成你所需要解决的任务的函数. 三个函数的复合满足结合律 所以 多个函数的复合满足广义结合律. 对于满足结合律的整数乘法,你常常要分解一个整数; 类似地,在这里你要分解一个函数, 分解一个整数的目的,也许是去获得一种, 更有利于乘法和除法运算的, 对这个整数的表示方式; 而当你以不同的方式分解一个函数,可能你就会对这个函数形成不同的理解方式, 所以我想, **分解一个函数的目的是为了形成对这个函数的最恰当的理解方式**. 当你以不同的方式分解一个函数, 可能 机器计算这个函数时 对时间和空间的消耗程度也不同, 所以 **分解一个函数的目的也可以是为了获得最优的计算速度或最节省内存**.

Forth的用户常把他们喜爱的Forth语言比喻为一个``放大器", 意思是说,好的编码者的能力会被放大,坏的编码者的愚蠢也会被放大.

[13]有一个Forth方言就是以``factor"命名的.

三染色算法与改进

下面是在 小语言学家 中的渐进式垃圾回收器,代码在四百行以内. 代码在前说明在后.

代码

the construction & clr, car, cdr

```
(* the construction of pair : [unit : byte]
 * clr:
 *     ||  1 : color     ||
 * car:
 *     ||  8 : type tag  ||
 *     ||  8 : value     ||
 * cdr:
 *     ||  8 : type tag  ||
 *     ||  8 : value     ||
 *)

(* the following constants are defined in assembler:
 * ConsBytesSize == 33
 * ClrBytesSize  ==  1
 * CarBytesSize  == 16
 * CdrBytesSize  == 16
 *)

: clr (* [address, <pair-like>] -- color-byte *)
  drop (* drop the type-tag *)
  sub1 fetchByte Exit
; defineFunction


: car (* [address, <pair-like>] -- [value, type] *)
  drop (* drop the type-tag *)
  fetchTwo Exit
; defineFunction

: cdr (* [address, <pair-like>] -- [value, type] *)
  drop (* drop the type-tag *)
  CarBytesSize + fetchTwo Exit
; defineFunction
```

color & set-clr!

```
0 : White ; defineConst
1 : Black ; defineConst
```

```
(* set three offsets used by fetchByte, setBit, clearBit *)
0 : VariableColorOffsetForFinding  ; defineVar
1 : VariableColorOffsetForMarking  ; defineVar
2 : VariableColorOffsetForCleaning ; defineVar

: ColorOffsetForFinding
  VariableColorOffsetForFinding
  fetch Exit
; defineFunction

: ColorOffsetForMarking
  VariableColorOffsetForMarking
  fetch Exit
; defineFunction

: ColorOffsetForCleaning
  VariableColorOffsetForCleaning
  fetch Exit
; defineFunction

: set-clr!
  (* [address, <pair-like>], color-byte --
     [address, <pair-like>] *)
  drop (* drop the type-tag *)
  x|over|xx sub1 saveByte Exit
; defineFunction
```

marking

```
: whiteColorForMarking? (* color-byte -- True or False *)
  ColorOffsetForMarking fetchBit White == Exit
; defineFunction

: blackColorForMarking? (* color-byte -- True or False *)
  ColorOffsetForMarking fetchBit Black == Exit
; defineFunction

: black-<pair>? (* [address, <pair-like>] -- True or False *)
  clr blackColorForMarking? Exit
; defineFunction

: try,white->grey
  (* [address, <pair-like>] -- [address, <pair-like>] *)
  twoDuplicate clr
  duplicate
  whiteColorForMarking? if
    ColorOffsetForMarking setBit set-clr!
    over pushGreyPairStack
    Exit
  then
  drop (* drop the color-byte *) Exit
; defineFunction
```

```
: one,try,grey->black (* -- *)
  emptyGreyPairStack? if
    Exit
  then
  popGreyPairStack duplicate
    fetchTwo duplicate <pair-like>? if
      try,white->grey
    then twoDrop
    CarBytesSize +
    fetchTwo duplicate <pair-like>? if
      try,white->grey
    then twoDrop
  Exit
; defineFunction


(* the following is a help-function of all,grey->black
 * the GreyPairStack must not be empty when it is called *)

: one,grey->black (* -- *)
  popGreyPairStack duplicate
    fetchTwo duplicate <pair-like>? if
      try,white->grey
    then twoDrop
    CarBytesSize +
    fetchTwo duplicate <pair-like>? if
      try,white->grey
    then twoDrop
  Exit
; defineFunction

: all,grey->black (* -- *)
  emptyGreyPairStack? if
    Exit
  then
  one,grey->black
  all,grey->black Exit
; defineRecursiveFunction
```

set!, set-car!, set-cdr!

```
: set!
  (* VarForTypedValue[address], [value, type] --
     VarForTypedValue[address] *)
  duplicate <pair-like>? if
  all,grey->black (* to be incremental-gc is to call this function here *)
    try,white->grey
  then
  x|over|xx saveTwo
  Exit
; defineFunction

: help,set-car!&set-cdr!,for-black-<pair>
  (* [valus, type] -- [valus, type] *)
```

```
  duplicate <pair-like>? if
   all,grey->black (* to be incremental-gc is to call this function here *)
    try,white->grey
  then
  Exit
; defineFunction

: set-car!
  (* [address, <pair-like>], [valus, type] --
     [address, <pair-like>] *)
  drop (* drop the type-tag overed *)
  xx|over|xx black-<pair>? if
    help,set-car!&set-cdr!,for-black-<pair>
  then
  x|over|xxx saveTwo
  Exit
; defineFunction

: set-cdr!
  (* [address, <pair-like>], [valus, type] --
     [address, <pair-like>] *)
  drop (* drop the type-tag overed *)
  xx|over|xx black-<pair>? if
    help,set-car!&set-cdr!,for-black-<pair>
  then
  x|over|xxx CarBytesSize + saveTwo
  Exit
; defineFunction
```

marking & define

```
(* recall
 * a word in the dictionary [unit : CellWidth = 8 bytes]
 *   ||  1 : name-string-header  ||
 *   ||  m : name-string  ||
 *   ||  1 : SizeOfFunctionBody  ||
 *   ||  1 : identification  ||
 *   ||  1 : link  ||
 *   ||  1 : type  ||
 *   ||  1 : address-of-name-string-header  ||
 *   ||  1 : address-of-explainer  ||
 *   ||  n : body  ||
 * where
 *   ||  1 : type  ||
 * ==
 *   | type-bit-63 | ... | type-bit-1 | type-bit-0 |
 * type-bit-0 is for HiddenWord
 * type-bit-1 is for VariableOfTypedValue
 *)

: createWordHeaderForTypedValue
  (* string[address, length] -- word[address of link] *)
  Here fetch xx|swap|x (* address-of-name-string-header *)
  appendStringToHere
```

```
  Here fetch appendNumberToHere (* identification *)
  Here fetch (* leave the word[link] *)
  Zero appendNumberToHere (* link *)
  Two appendNumberToHere  (* type *)
  swap
  appendNumberToHere (* address-of-name-string-header *)
  Exit
; defineFunction


: define
 (* [value, type], wordList[address, the number of words] -- *)
  tailAndHeadOfWordList
  createWordHeaderForTypedValue
  VarExplainer appendNumberToHere
  xx|swap|x twoDrop xx|swap|x  (* leave wordHeader *)
  duplicate <pair-like>? if
    try,white->grey
  then
  appendNumberToHere appendNumberToHere
  addNewWordToDictionary
  Exit
; defineFunction
```

finding & cons(<pair>)

```
(* the following functions are helping cons *)

: clearColorBitOfPairForCleaning
  (* pair[address] -- pair[address] *)
  <pair>
  twoDuplicate clr
  ColorOffsetForCleaning clearBit
  set-clr!
  drop
  Exit
; defineFunction

: whiteColorForFinding? (* color-byte -- True or False *)
  ColorOffsetForFinding fetchBit White ==
  Exit
; defineFunction

: findNextFreePairConstruction
 (* pair[address] -- Zero or NextFreePairConstruction[address] *)
  duplicate LastPairConstruction == if
    drop Zero Exit
  then
  ConsBytesSize +
  clearColorBitOfPairForCleaning
  duplicate <pair> clr
  whiteColorForFinding? if
    Exit
  then
```

```
  findNextFreePairConstruction
  Exit
; defineRecursiveFunction


(* VariableColorOffsetForFinding  --> VariableColorOffsetForCleaning
 * VariableColorOffsetForMarking  --> VariableColorOffsetForFinding
 * VariableColorOffsetForCleaning --> VariableColorOffsetForMarking *)
: resetColorOffsets (* -- *)
  VariableColorOffsetForFinding  fetch
  VariableColorOffsetForMarking  fetch
  VariableColorOffsetForCleaning fetch
  VariableColorOffsetForMarking  save
  VariableColorOffsetForFinding  save
  VariableColorOffsetForCleaning save
  Exit
; defineFunction


: dynamicVariableWordFor<pair-like>?
  (* word[address of link] -- True or False *)
  duplicate dynamicVariableWord? if
  wordLinkToWordExplainer execute fetchTwo
  swap drop
  <pair-like>?
  Exit
  then
  drop False Exit
; defineFunction

: help,pushAllRootNodeIntoGreyPairStack
  (* word[address of link] -- *)
  duplicate lastWordInTheDictionary? if
    drop Exit
  then
  duplicate dynamicVariableWordFor<pair-like>? if
  duplicate wordLinkToWordExplainer execute fetchTwo
  try,white->grey twoDrop
  then
  nextWordInTheDictionary
  help,pushAllRootNodeIntoGreyPairStack
  Exit
; defineRecursiveFunction

: pushAllRootNodeIntoGreyPairStack (* -- *)
  FirstWordInDictionary fetch
  help,pushAllRootNodeIntoGreyPairStack
  Exit
; defineFunction


: addressOfPair? (* value -- Ture or False *)
  duplicate FirstPairConstruction < if
    drop False Exit
  then
  duplicate LastPairConstruction > if
```

```
    drop False Exit
  then
  FirstPairConstruction - ConsBytesSize mod zero?
  Exit
; defineFunction


: help,allPairsInArgumtStack,try,white->grey
  (* address of a Cell in ArgumtStack -- *)
  duplicate ArgumtStackTop > if
    drop Exit
  then
  duplicate fetch addressOfPair? if
    duplicate fetch
    <pair> try,white->grey
    twoDrop
  then
  CellWidth +
  help,allPairsInArgumtStack,try,white->grey
  Exit
; defineRecursiveFunction

(* tryToDarkenAllWhirtPairsInArgumtStack *)
: allPairsInArgumtStack,try,white->grey (* -- *)
  fetchArgumtStackPointer
  help,allPairsInArgumtStack,try,white->grey
  Exit
; defineFunction

: resetVariablesAboutString (* -- *)
 CurrFreeStringAddress,to fetch CurrFreeStringAddress,from save
 Variable,StringHeap,from fetch CurrFreeStringAddress,to  save
 Variable,StringHeap,to  fetch Variable,StringHeap,from  save
 CurrFreeStringAddress,to fetch Variable,StringHeap,to   save
  Exit
; defineFunction

:" cons said: ``Memory for cons is used up! No value is returned!''"
: String,cons,MemoryIsUsedUp ; defineConstString

: cons (* -- [address, <pair>] *)
 CurrFreePairConstruction fetch <pair> (* leave the return <value> *)
 CurrFreePairConstruction fetch findNextFreePairConstruction
  duplicate notZero? (* Zero denotes fail to find *) if
    CurrFreePairConstruction save
    Exit
 then drop (* drop the Zero, which denotes fail to find, need gc *)
  allPairsInArgumtStack,try,white->grey
  all,grey->black
 resetColorOffsets resetVariablesAboutString (* note the timing to reset *)
  pushAllRootNodeIntoGreyPairStack
 InFrontOfTheFirstPairConstruction findNextFreePairConstruction
  duplicate notZero? if
    CurrFreePairConstruction save
    Exit
  then drop
```

```
    (* if after gc still fail to find, we know the memory is used up *)
     twoDrop
     String,cons,MemoryIsUsedUp printString cr
     debuger
     Exit
   ; defineFunction
```

对代码的说明

为了重新利用一个曾经被分配过的节点, 我需要证明, **这个节点已经不会被当前的任何全局变量或局部变量所引用到了**.

mark-and-sweep算法通过``标记所有能够被引用到的节点", 来知道哪些节点是不能被引用到的.

Dijkstra的三染色算法的想法很简单:古典的mark-and-sweep算法, 会深度优先地遍历一个特殊的有向图(如前所述, 这种图非常像二叉树,但是可以形成特殊的圈); 在遍历的过程中会将很多返回点入栈(这个栈将被称为GreyPairStack), 就像ReturnStack中保存的函数返回点代表了当前的计算状态一样, 这里栈中保存的节点也代表了marking所进行到的状态; 把这些在栈中的节点视为是灰色的, 就得到了三染色算法.

因为有GreyPairStack来保存染色的状态, 所以染色的过程可以在任何时候进行; 两个能够对pair形成副作用的函数`set-car!` `set-cdr!`, 可以随时把新的点添加到GreyPairStack中.

只要GreyPairStack是空的, 那么所有的染色就进行完了. 所以只要当所有自由的pair都分配完毕之时, 把GreyPairStack中的所有节点都处理完, 染色过程就结束了.

渐进性就在于, **把需要集中进行的mark-and-sweep分散到别的地方**. 上面的Dijkstra所介绍的技巧把marking分散到了别的地方; 而我的改进把sweep分散到了别的地方(表面上看是消除了sweep的过程, 其实是把计算分散了).

技巧在于,把sweep的过程改成find-and-clean, 利用三个颜色(黑白灰)的同时, 也在每个pair上设置三个染色位置, 一个为marking,一个为finding,一个为cleaning.

每个gc的每个工作周期中, finding会看专门为它准备的染色位置,来决定那个pair是自由的; finding一定会扫描一遍所有pair所组成的数组, 所以让只要cleaning跟着finding一起工作, 那么在一个工作周期结束后finding一定会清理完所有需要被它清理染色位置; marking的工作与find-and-clean无关并且互不影响, marking可以随时把一个白点染灰,或把一个灰点染黑.

每次gc的一个工作周期结束之后, 只要对三个染色位置的offset做一个三周期的置换:

```
ColorOffsetForFinding   -->  ColorOffsetForCleaning
ColorOffsetForMarking   -->  ColorOffsetForFinding
ColorOffsetForCleaning  -->  ColorOffsetForMarking
```

下一个工作周期中, marking所使用的就是被上一个工作周期中cleaning所清理干净的染色位置; finding所使用的就是被上一个工作周期中marking所染色好了的染色位置; cleaning所清理的就是上一个工作周期中finding所使用过的染色位置.

引用

- John McCarthy, ``Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", MIT. 1960.
- Charles H. Moore, ``programming a problem-oriented-language", unpublished book, 1970; posted on Moore's website by himself, 2011.
- Daniel P. Friedman; Matthias Felleisen, ``The Little Schemer", Fourth Edition, 1996.
- Henk Barendregt; Erik Barendsen, ``Introduction to Lambda Calculus", 2000.
- Henk Barendregt, ``The Lambda Calculus, Its syntax and semantics", 1980.
- Henk Barendregt, ``The impact of the lambda calculus in logic and computer science", Computing Science Institute, Nijmegen University The Netherlands, 1997.
- von Thun. Manfred; Thomas. Reuben, ``Joy: Forth's Functional Cousin", Proceedings of the 17th EuroForth Conference, 2001.
- Andrew W. Appel, ``Garbage Collection Can Be Faster Than Stack Allocation", Department of Computer Science, 1987.
- Robert Hieb; R. Kent Dybvig; Carl Bruggeman, ``Representing Control in the Presence of First-Class Continuations", Indiana University, Computer Science Department, 1990.
- Edsgar W. Dijkstra; Leslie Lamport; A. J. Martin; C. S. Scholten; and E. F. M. Steffens, ``On-the-fly garbage collection: An exercise in cooperation", In Lecture Notes in Computer Science, No. 46. Springer-Verlag, 1976.
- H. Abelson; D. P. Friedman; R. K. Dybvig; G. J. Sussman; et al. ``Revised Revised Revised Revised Revised Report on the Algorithmic Language Scheme", 1998