A General Framework for Certifying Garbage Collectors and Their Mutators (Supplement)

Andrew McCreight[†] Zhong Shao[†] Chunxiao Lin[‡] Long Li[‡]

[†]Department of Computer Science
Yale University
New Haven, CT 06520-8285, U.S.A.
{aem, shao}@cs.yale.edu

[‡]Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui 230026, China
{cxlin3, liwis}@mail.ustc.edu.cn

1. Introduction

This is a supplement to the main paper (available from [5]), containing details about the implementation that were omitted for space reasons. In Section 2, we give the mutator view properties we require. In Sections 3 and 4 we give the pseudocode and assembly code for the collectors described in our paper.

2. Additional mutator view properties

In order to allow the mutator to manipulate state containing a mutator view, we require that *mview* satisfies some basic properties. These are:

1. The concrete representation of an atomic value is the same as its virtual representation.
2. Registers not in the domain of the virtual state (excepting gcInfo) can be changed in the concrete state without affecting the view.
3. Registers in the domain of the virtual state can be copied to other registers (thereby expanding the domain of the virtual state).
4. Registers can be removed from the domain of the virtual state.
5. Registers with atomic values (once again, excluding gcInfo) can be added to the domain of the virtual state.

3. Pseudocode

In this section, we present pseudo code for mark-sweep [3], Cheney [2, 3, 1] and Baker [4, 3] garbage collectors.

3.1 Mark-sweep collector

```
// start and end of object heap
word *hpStart, *hpEnd;

// bottom and top of mark stack
word *stBot, *stTop;
// end of mark stack buffer
word* stBuff;

markField (word* x) {
  // return if x is atomic,
  //   or has already been marked
  if (((x & 1) <> 0) || (x[-1] == 1))
    return;
  // mark x, push x on stack
  x[-1] = 1;
  stackPush(x);
}

mark (word* root) {
  // initialize stack
  stTop = stBot;

  markField(root);
  while (! stackEmpty()) {
    x = stackPop();
    mark(x[0]);
    mark(x[1]);
  }
}

sweep () {
  // start sweep at first object
  sweep = hpStart + 1;
  free = NULL;

  // sweep the entire heap
  while (sweep + 2 <= hpEnd) {
    if (sweep[-1] == 0) {
      sweep[1] = free;
      free = sweep;
    } else {
      sweep[-1] = 0;
    }
    sweep = sweep + 3;
  }

}

stackPush (x) {
  if (stBuff <= stTop) {
    while (true) {}
  }
  stTop[0] = x;
  stTop += 1;
}

// precondition: stack is not empty
stackPop () {
  stTop = stTop - 4;
  return stTop[0];
```

```
}

stackEmpty () {
  return (stBot == stTop);
}
```

3.2 Cheney collector

```
// pointer to start and end of from-space
word *frStart, *frEnd;

// pointer to start and end of to-space
word *toStart, *toEnd;

// pointer to first free object
word *free;

// pointer to object being scanned
word *scan;

bool fromSpacePtr (word* ptr) {
  if (ptr & 1 == 1) return false;
  return !(ptr < frStart || ptr >= frEnd);
}

bool toSpacePtr (word* ptr) {
  if (ptr & 1 == 1) return false;
  return !(ptr < toStart || ptr >= toEnd);
}

word* fwdObj(word* obj) {
  // copy fields from obj to free
  free[0] = obj[0];
  free[1] = obj[1];
  // store forwarding pointer in obj
  obj[0] = free;
  // new object is at free
  newObj = free;
  free += 8;
  return newObj;
}

scanField (word* fieldPtr) {
  fval = *fieldPtr;
  if (fval && 1  == 0) { // if fval is a pointer
    field1 = fval[0];
    if (toSpacePtr(field1)) {
      // if 1st field is to space ptr, object
      // has already been forwarded
      *fieldPtr = field1;
    } else {
      *fieldPtr = fwdObj(fval);
    }
  }
}

cheneyGC (word* root) {
  // switch semi-spaces
  swap(frStart, toStart);
  swap(frEnd, toEnd);

  // initialize free and scan pointers
  free = scan = toStart;

  // make sure the root is copied
  scanField(root);

  // scan all to-space objects
  while (scan != free) {
    scanField(scan);
    scanField(4 + scan);
    scan += 8;
  }

}

alloc (word* root) {
  if (free == toEnd) {
    // out of space: call collector
    cheneyGC(root);
    alloc(root);
  }
  free[0] = NULL;
  free[1] = NULL;
  temp = free;
  free += 8;
  return temp;
}
```

3.3 Baker

```
// same as in the Cheney collector
word *frStart, *frEnd, *toStart, *toEnd;
word *scan, *free;

// points to the end of the free space,
// because we allocate mutator objects
// from the end
word *alloc;

word* fwdObj(word* obj) {
  // make sure we have enough space
  if (free == alloc) abort();

  // rest is same as Cheney
  free[0] = obj[0];
  free[1] = obj[1];
  obj[0] = free;
  newObj = free;
  free += 8;
  return newObj;
}

scanField (word* fieldPtr) {
  fval = *fieldPtr;
  if (fromSpacePtr(fval)) {
    field1 = fval[0];
    if (toSpacePtr(field1))
      *fieldPtr = field1;
    else
      *fieldPtr = fwdObj(fval);
  }
}

bakerGC(word* root) {
  count = 0;

  if (free == alloc) {
    if (scan < free)
      abort(); // we didn't finish scanning

    // flip spaces, scan root
    swap(frStart, toStart);
    swap(frEnd, toEnd);
    free = scan = toStart;
    alloc = toEnd;
```

```
    bakerScanField(root);
  }

  while (scan != free &&
         count < scan_per_gc) {
    bakerScanField(scan);
    bakerScanField(scan+4);
    scan = scan + 8;
    count = count + 1;
  }

}
```

4. Assembly Code

In this section, we give the actual assembly code for the GCs we discussed in this paper. The implementation of the copying collectors differs slightly from the pseudo code because we used the negation of the semi-space tests to simplify the implementation, though in hindsight this isn't that helpful. This has only a minor effect on the code.

4.1 Mark-sweep collector

4.1.1 Mark

```
MARKFIELD:
                addiu rcst3,ra,0;
                andi r31,a0,1;
                bne r31, zero, RETURN3;
                addiu r31,zero,4;
                subu a0,a0,r31;
                lw ast,0(a0);
                addiu r31,zero,1;
                beq ast, r31, RETURN3;
                sw r31,0(a0);
                addiu a0,a0,4;
                jal STACKPUSH, RETURN3

MARK:
                addiu rcst2,ra,0;
                lw rbot,12(rgcInfo);
                lw rbuff,16(rgcInfo);
                addiu rtop,rbot,0;
                addiu a0,rroot,0;
                jal MARKFIELD, MARKLOOP

MARKLOOP:
                jal STACKEMPTY, MARKLOOPBODY

MARKLOOPBODY:
                bne v0, zero, RETURN2;
                jal STACKPOP, MARKFIRST

MARKFIRST:
                lw a0,0(v0);
                jal MARKFIELD, MARKSECOND

MARKSECOND:
                lw a0,4(v0);
                jal MARKFIELD, MARKLOOP
```

4.1.2 Sweep

```
SWEEP:
                lw rsweep,0(rgcInfo);
                addiu rsweep,rsweep,4;
                lw rhpEnd,4(rgcInfo);
                addiu rfree,zero,0;
                j SWEEPLOOP

SWEEPLOOP:
                addiu ast,rsweep,8;
                sltu ast,rhpEnd,ast;
                bne ast, zero, SWEEPFIN;
                addiu ast,zero,4;
                subu ast,rsweep,ast;
                lw v0,0(ast);
                beq v0, zero, SWEEPADD;
                sw zero,0(ast);
                j SWEEPNEXT

SWEEPADD:
                sw rfree,4(rsweep);
                addiu rfree,rsweep,0;
                j SWEEPNEXT

SWEEPNEXT:
                addiu rsweep,rsweep,12;
                j SWEEPLOOP

SWEEPFIN:
                sw rfree,8(rgcInfo);
                jr ra
```

4.1.3 Top level

```
MARKSWEEPGC:
                addiu rcst1,ra,0;
                jal MARK, MARKSWEEPSWEEP

MARKSWEEPSWEEP:
                jal SWEEP, RETURN1
```

4.1.4 Stack code

```
ISEMPTY:
            beq rbot, rtop, STACKEMPTYT;
            addiu v0,zero,0;
            jr ra

ISEMPTYT:
            addiu v0,zero,1;
            jr ra

PUSH:
            addiu rtemp,rtop,4;
            sltu rtemp,rbuff,rtemp;
            bne rtemp, zero, INFLOOP;
            sw a0,0(rtop);
            addiu rtop,rtop,4;
            jr ra

POP:
            addiu v0,zero,4;
            subu rtop,rtop,v0;
            lw v0,0(rtop);
            jr ra
```

4.1.5 Misc

```
RETURN1:
        addiu ra,rcst1,0;
        jr ra

RETURN2:
        addiu ra,rcst2,0;
        jr ra

RETURN3:
        addiu ra,rcst3,0;
        jr ra
```

4.2 Cheney collector

4.2.1 Main collector code

```
CHLOOP:
        beq rscan, rfree, CHRET
        addiu a0,rscan,0
        jal CHSCANF, CHLOOP2

CHLOOP2:
        addiu a0,rscan,4
        addiu rscan,rscan,8
        jal CHSCANF, CHLOOP

CHRET:
        addiu ra,raSave,0
        jr ra

CHLOOPH:
        lw rroot,12(rgcInfo)
        sw rtoEnd,12(rgcInfo)
        j CHLOOP

CHENTER:
        lw rtoStart,0(rgcInfo)
        lw ast,8(rgcInfo)
        sw ast,0(rgcInfo)
        sw rtoStart,8(rgcInfo)
        lw rtoEnd,4(rgcInfo)
        lw ast,12(rgcInfo)
        sw ast,4(rgcInfo)
        addiu rfree,rtoStart,0
        addiu rscan,rtoStart,0
        addiu raSave,ra,0
        sw rroot,12(rgcInfo)
        addiu a0,rgcInfo,12
        jal CHSCANF, CHLOOPH
```

4.2.2 Allocator

```
CHALLOC:
        lw rfree,16(rgcInfo)
        lw ast,12(rgcInfo)
        bne rfree, ast, CHALLOC2
        jal CHENTER, CHALLOC

CHALLOC2:
        addiu v0,zero,1
        sw v0,0(rfree)
        sw v0,4(rfree)
        addiu v0,rfree,0
        addiu rfree,rfree,8
        sw rfree,16(rgcInfo)
        jr ra
```

4.2.3 Scan field

```
CHSCANF:
        lw t1,0(a0);
        andi ast,t1,1;
        bne ast, zero, return;
        addiu t0,a0,0;
        lw t2,0(t1);
        addiu t7,ra,0;
        addiu t4,rfree,0;
        addiu a0,t2,0;
        jal NOT_TO_PTR, CHSCANP

CHSCANP:
        addiu rfree,t4,0;
        beq v0, zero, CHNOCOPY;
        addiu a0,t1,0;
        jal CHFWDOBJ, CHCOPIED

CHNOCOPY:
        sw t2,0(t0);
        addiu ra,t7,0;
        jr ra

CHCOPIED:
        sw v0,0(t0);
        addiu ra,t7,0;
        jr ra
```

4.2.4 Forwarding

```
CHFWDOBJ:
        lw ast,0(a0);
        sw ast,0(rfree);
        lw ast,4(a0);
        sw ast,4(rfree);
        sw rfree,0(a0);
        addiu v0,rfree,0;
        addiu rfree,rfree,8;
        jr ra
```

4.3 Baker collector

```
BSCANFIELD:
        lw s1,0(a0);
        addiu s0,a0,0;
        addiu s7,ra,0;
        addiu s4,rfree,0;
        addiu a0,s1,0;
        jal NOT_FROM_PTR, BSCANFIELD2

BSCANFIELD2:
        bne v0, zero, RRETURN;
        lw s2,0(s1);
        addiu a0,s2,0;
        jal NOT_TO_PTR, SCANPOINTER

SCANPOINTER:
        addiu rfree,s4,0;
        beq v0, zero, SCANNOCOPY;
        addiu a0,s1,0;
        jal BFWDOBJ, SCANCOPIED

SCANNOCOPY:
        sw s2,0(s0);
        addiu ra,s7,0;
        jr ra

SCANCOPIED:
        sw v0,0(s0);
        addiu ra,s7,0;
        jr ra

RRETURN:
        addiu ra,s7,0;
        jr ra
```

```
BAKERENTER:
        lw rfrStart,0(rgcInfo);
        lw rfrEnd,4(rgcInfo);
        lw rtoStart,8(rgcInfo);
        lw rtoEnd,12(rgcInfo);
        lw rscan,16(rgcInfo);
        lw rfree,20(rgcInfo);
        lw ralloc,24(rgcInfo);
        addiu rcount,zero,0;
        addiu raSave,ra,0;

        bne rfree, ralloc, BAKERLOOP;

        sltu ast,rscan,rfree;
        bne ast, zero, INFLOOP;

        addiu ast,rfrStart,0;
        addiu rfrStart,rtoStart,0;
        addiu rtoStart,ast,0;
        addiu ast,rfrEnd,0;
        addiu rfrEnd,rtoEnd,0;
        addiu rtoEnd,ast,0;

        sw rfrStart,0(rgcInfo);
        sw rfrEnd,4(rgcInfo);
        sw rtoStart,8(rgcInfo);
        sw rtoEnd,12(rgcInfo);

        addiu rfree,rtoStart,0;
        addiu rscan,rtoStart,0;
        addiu ralloc,rtoEnd,0;

        sw rroot,16(rgcInfo);
        addiu a0,rgcInfo,16;
        jal BSCANFIELD, RESTOREROOT

RESTOREROOT:
        lw rroot,16(rgcInfo);
        j BAKERLOOP
BAKERLOOP:
        beq rscan, rfree, BAKEREXIT;

        addiu ast,zero,scan_per_gc;
        sltu ast,rcount,ast;
        beq ast, zero, BAKEREXIT;

        addiu a0,rscan,0;
        jal BSCANFIELD, BAKERLOOP2

BAKERLOOP2:
        addiu a0,rscan,4;
        addiu rscan,rscan,8;
        addiu rcount,rcount,1;
        jal BSCANFIELD, BAKERLOOP

BAKEREXIT:
        addiu ra,raSave,0;
        sw rscan,16(rgcInfo);
        sw rfree,20(rgcInfo);
        sw ralloc,24(rgcInfo);
        jr ra
INFLOOP:
    j INFLOOP
```

```
NOT_FROM_PTR:
                andi v0,a0,1;
                bne v0, zero, RETURN;
                sltu v0,a0,rfrStart;
                bne v0, zero, RETURN;
                addiu v0,a0,1;
                sltu v0,rfrEnd,v0;
                bne v0, zero, RETURN;
                addiu v0,zero,0;
                jr ra

NOT_TO_PTR:
                andi v0,a0,1;
                bne v0, zero, RETURN;
                sltu v0,a0,rtoStart;
                bne v0, zero, RETURN;
                addiu v0,a0,1;
                sltu v0,rtoEnd,v0;
                bne v0, zero, RETURN;
                addiu v0,zero,0;
                jr ra

RETURN:
                jr ra
BFWDOBJ:
        beq rfree, ralloc, INFLOOP;
        lw ast,0(a0);
        sw ast,0(rfree);
        lw ast,4(a0);
        sw ast,4(rfree);
        sw rfree,0(a0);
        addiu v0,rfree,0;
        addiu rfree,rfree,8;
        jr ra
```

References

[1] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *POPL '04: Proc. of the 31st ACM SIGPLAN-SIGACT symp. on Principles of prog. lang.*, pages 220–231, New York, NY, USA, 2004. ACM Press.

[2] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[3] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[4] H. G. B. Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[5] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators (extended version), and coq implementation. `http://flint.cs.yale.edu/flint/publications/hgc.html`.