```
/*
 * This is going to be a version of the copying garbage collector for use
 * on 64−bit machines when it has just loaded a 32−bit image file.
 * its job is then to do a copying−style garbage collection where the
 * source space is set up to be in 32−bit format and the destination
 * is in 64−bit form! One nasty issue is that of forwarding addresses, which
 * can no longer be normal native references − in the 32−bit space ALL
 * addresses will have to live in a fort of segmented form
 *    −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 *   | <page number> | <offset within page> | <tags> |
 *    −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
 * whh=ich is the form that the have while in an image file.
 */

static int trailing_heap_pages_count,
           trailing_vheap_pages_count;

typedef int32_t Source_Object;
typedef Lisp_Object Destination_Object;

/*
 * This is going to be "just" the code from the regular garbage collector
 * adjusted so that the source space is in smaller items. Well perhaps if I
 * was clever enough I could make it such that it just had one type for its
 * source and another for its destination half−space and one bit of
 * code here could copy either preserving, widening or narrowing
 * representation.
 */

static void copy(Source_Object *p)
/*
 * This copies the object pointed at by p from the old to the new semi−space,
 * and returns a copy to the pointer.  If scans the copied material to copy
 * all relevent sub−structures to the new semi−space.
 */
{
    Lisp_Object nil = C_nil;
    char *fr = (char *)fringe, *vfr = (char *)vfringe;
    char *tr_fr = fr, *tr_vfr = vfr;
    void *p1;
#define CONT           0
#define DONE_CAR      −1
#define DONE_VALUE    −2
#define DONE_ENV      −3
#define DONE_PNAME    −4
#define DONE_PLIST    −5
#define DONE_FASTGETS −6
    int next = CONT;
    char *tr=NULL;
#ifdef DEBUG_GC
    term_printf("Copy [%p] %p\n", (void *)p, (void *)*p);
#endif
/*
 * The code here is a simulation of multiple procedure calls to the
 * code that copies a single object.  What might otherwise have been
 * a "return address" in the calls is handled by the variable "next" which
 * takes positive values while copying vectors, and negative ones in
 * the more common cases. I use "for (;;)" blocks a lot so that I can
 * use "break" and "continue" to leap around in the code − maybe I
 * would do better to be honest and use regular labels and "goto"
 * statements.
 */
    for (;;)
    {
/*
 * Copy one object, pointed at by p, from the old semi−space into the new
 * one.
 */
        Lisp_Object a = *p;
#ifdef DEBUG_GC
    term_printf("Next copy [%p] %p\n", (void *)p, (void *)*p);
#endif
        for (;;)
        {
            if (a == nil) break;    /* common and cheap enough to test here */
            else if (is_immed_or_cons(a))
            {   if (is_cons(a))
                {
                    Lisp_Object w;
                    w = qcar(a);
                    if (is_cons(w) && is_marked_p(w)) /* a forwarding address */
                    {   *p = flip_mark_bit_p(w);
                        break;
                    }
                    fr = fr − sizeof(Cons_Cell);
                    cons_cells += 2*CELL;
/*
 * When I am doing regular calculation I leave myself a bunch of spare
 * words (size SPARE bytes) so that I can afford to do several cons operations
 * between tests.  Here I do careful tests on every step, and so I can
 * sail much closer to the wind wrt filling up space.
 */
                    if (fr <= (char *)heaplimit − SPARE + 32)
                    {   char *hl = (char *)heaplimit;
                        void *p;
                        uintptr_t len = (uintptr_t)(fr − (hl − SPARE) +
                                                     sizeof(Cons_Cell));
                        car32(hl − SPARE) = len;
                        qcar(fr) = SPID_GCMARK;
                        if (pages_count == 0)
                        {   term_printf("pages_count = 0 in GC\n");
                            ensure_screen();
                            abort();
                            return;
                        }
                        p = pages[−−pages_count];
                        zero_out(p);
                        new_heap_pages[new_heap_pages_count++] = p;
                        heaplimit = quadword_align_up((intptr_t)p);
                        hl = (char *)heaplimit;
                        car32(heaplimit) = CSL_PAGE_SIZE;
                        fr = hl + CSL_PAGE_SIZE − sizeof(Cons_Cell);
                        heaplimit = (Lisp_Object)(hl + SPARE);
                    }
                    qcar(fr) = w;
                    qcdr(fr) = qcdr(a);
                    *p = w = (Lisp_Object)(fr + TAG_CONS);
                    qcar(a) = flip_mark_bit_p(w);
                    break;
                }
                else if (is_bps(a))
                {   char *d = data_of_bps(a) − CELL, *rr;
                    intptr_t alloc_size;
                    Header h = *(Header *)d;
                    intptr_t len;
                    if (is_bps(h))  /* Replacement handle in header field? */
                    {   *p = h ;
                        break;
                    }
                    len = length_of_header(h);
                    alloc_size = (intptr_t)doubleword_align_up(len);
                    bytestreams += alloc_size;
                    for (;;)
                    {   char *cf = (char *)codefringe,
                             *cl = (char *)codelimit;
                        uintptr_t free = (uintptr_t)(cf − cl);
                        if (alloc_size > (intptr_t)free)
                        {
                            void *p;
                            if (codelimit != 0)
                            {   uintptr_t len = (uintptr_t)(cf − (cl − 8));
                                car32(cl − 8) = len;
                            }
                            if (pages_count == 0)
                            {   term_printf("pages_count = 0 in GC\n");
                                ensure_screen();
                                abort();
                                return;
                            }
                            p = pages[−−pages_count];
                            zero_out(p);
                            new_bps_pages[new_bps_pages_count++] = p;
                            cl = (char *)doubleword_align_up((intptr_t)p);
                            codefringe = (Lisp_Object)(cl + CSL_PAGE_SIZE);
                            codelimit = (Lisp_Object)(cl + 8);
                            continue;
                        }
                        rr = cf − alloc_size;
                        codefringe = (Lisp_Object)rr;
/*
 * See comments in fns2.c for the curious packing here!
 */
                        *(Header *)d = *p = TAG_BPS +
                           (((intptr_t)((rr + CELL) − (cl − 8)) &
                             (PAGE_POWER_OF_TWO−4)) << 6) +
                           (((intptr_t)(new_bps_pages_count−1))<<(PAGE_BITS+6));
                        /* Wow! How obscure!! */
                        *(Header *)rr = h;
                        memcpy(rr+CELL, d+CELL, alloc_size−CELL);
                        break;
                    }
                    break;
                }
                else break;        /* Immediate data drops out here */
            }
            else                    /* Here I have a symbol or vector */
            {   Header h;
                int tag;
                intptr_t len;
                tag = ((int)a) & TAG_BITS;
                a = (Lisp_Object)((char *)a - tag);
                h = *(Header *)a;
#ifdef DEBUG_GC
                term_printf("Header is %p\n", (void *)h);
#endif
                if (!is_odds(h))
                {   *p = h;
                    break;
                }
                if (tag == TAG_SYMBOL)
                    len = symhdr_length, symbol_heads += symhdr_length;
                else
                {   len = doubleword_align_up(length_of_header(h));
                    switch (type_of_header(h))
                    {
                case TYPE_STRING:
                        strings += len; break;
                case TYPE_BIGNUM:
                        big_numbers += len; break;
#ifdef COMMON
                case TYPE_SINGLE_FLOAT:
                case TYPE_LONG_FLOAT:
#endif
                case TYPE_DOUBLE_FLOAT:
                        box_floats += len; break;
                case TYPE_SIMPLE_VEC:
                        user_vectors += len; break;
                default:
                        other_mem += len; break;
                    }
                }
                for (;;)
                {   char *vl = (char *)vheaplimit;
                    uintptr_t free = (uintptr_t)(vl - vfr);
                    if (len > (intptr_t)free)
                    {   uintptr_t free1 =
                            (uintptr_t)(vfr - (vl - (CSL_PAGE_SIZE - 8)));
                        car32(vl - (CSL_PAGE_SIZE - 8)) = free1;
                        qcar(vfr) = 0;          /* sentinel value */
                        if (pages_count == 0)
                        {   term_printf("pages_count = 0 in GC\n");
                            ensure_screen();
                            abort();
                            return;
                        }
                        p1 = pages[--pages_count];
                        zero_out(p1);
                        new_vheap_pages[new_vheap_pages_count++] = p1;
                        vfr = (char *)doubleword_align_up((intptr_t)p1) + 8;
                        vl = vfr + (CSL_PAGE_SIZE - 16);
                        vheaplimit = (Lisp_Object)vl;
                        free1 = (uintptr_t)(vfr - (vl - (CSL_PAGE_SIZE - 8)));
                        car32(vl - (CSL_PAGE_SIZE - 8)) = free1;
                        continue;
                    }
                    *(Lisp_Object *)a = *p = (Lisp_Object)(vfr + tag);
                    *(Header *)vfr = h;
                    memcpy((char *)vfr+CELL, (char *)a+CELL, len-CELL);
                    vfr += len;
                    break;
                }
                break;
            }
        }
/*
 * Now I have copied one object - the next thing to do is to scan to see
 * if any further items are in the new space, and if so I will copy
 * their offspring.
 */
        for (;;)
        {
            switch (next)
            {
        case CONT:
                if (tr_fr != fr)
                {   tr_fr = tr_fr - sizeof(Cons_Cell);
                    if (qcar(tr_fr) == SPID_GCMARK)
                    {   char *w;
                        p1 = new_heap_pages[trailing_heap_pages_count++];
                        w = (char *)quadword_align_up((intptr_t)p1);
                        tr_fr = w + (CSL_PAGE_SIZE - sizeof(Cons_Cell));
                    }
                    next = DONE_CAR;
                    p = &qcar(tr_fr);
                    break;              /* Takes me to the outer loop */
                }
                else if (tr_vfr != vfr)
                {   Header h;
                    h = *(Header *)tr_vfr;
                    if (h == 0)
                    {   char *w;
                        p1 = new_vheap_pages[trailing_vheap_pages_count++];
                        w = (char *)doubleword_align_up((intptr_t)p1);
                        tr_vfr = w + 8;
                        h = *(Header *)tr_vfr;
                    }
                    if (is_symbol_header(h))
                    {   next = DONE_VALUE;
                        p = &(((Symbol_Head *)tr_vfr)->value);
                        break;
                    }
                    else
                    {   intptr_t len = doubleword_align_up(length_of_header(h));
                        tr = tr_vfr;
                        tr_vfr = tr_vfr + len;
```

```
                            switch (type_of_header(h))
                            {
#ifdef COMMON
                        case TYPE_SINGLE_FLOAT:
                        case TYPE_LONG_FLOAT:
#endif
                        case TYPE_DOUBLE_FLOAT:
                        case TYPE_BIGNUM:
                                continue;
                        case TYPE_MIXED1: case TYPE_MIXED2:
                        case TYPE_MIXED3: case TYPE_STREAM:
                                next = 2*CELL;
                                break;
/*
 * There is a slight delight here. The test "vector_holds_binary" is only
 * applicable if the header to be checked is a header of a genuine vector,
 * ie something that would have TAG_VECTOR in the pointer to it. But here
 * various numeric data types also live in the vector heap, so I need to
 * separate them out explicitly. The switch block here does slightly more than
 * it actually HAS to, since the vector_holds_binary test would happen to
 * deal with several of the numeric types "by accident", but I feel that
 * the security of listing them as separate cases is more important than the
 * minor speed−up that might come from exploiting such marginal behaviour.
 */
                        default:
                                if (vector_holds_binary(h)) continue;
#ifdef COMMON
                        case TYPE_RATNUM:
                        case TYPE_COMPLEX_NUM:
#endif
                                next = len − 2*CELL;
                                break;
                            }
                            p = (Lisp_Object *)(tr + next + CELL);
                            break;
                        }
                    }
                    else
                    {   fringe = (Lisp_Object)fr;
                        vfringe = (Lisp_Object)vfr;
                        return;         /* Final exit when all has been copied */
                    }
        case DONE_CAR:
                    next = CONT;
                    p = &qcdr(tr_fr);
                    break;
        case DONE_VALUE:
                    next = DONE_ENV;
                    p = &(((Symbol_Head *)tr_vfr)−>env);
                    break;
        case DONE_ENV:
                    next = DONE_FASTGETS;
                    p = &(((Symbol_Head *)tr_vfr)−>fastgets);
                    break;
        case DONE_FASTGETS:
                    next = DONE_PNAME;
                    p = &(((Symbol_Head *)tr_vfr)−>pname);
```

```
                break;
        case DONE_PNAME:
#ifndef COMMON
                next = CONT;
                p = &(((Symbol_Head *)tr_vfr)->plist);
                tr_vfr = tr_vfr + symhdr_length;
                break;
#else
                next = DONE_PLIST;
                p = &(((Symbol_Head *)tr_vfr)->plist);
                break;
        case DONE_PLIST:
                next = CONT;
                p = &(((Symbol_Head *)tr_vfr)->package);
                tr_vfr = tr_vfr + symhdr_length;
                break;
#endif
        default:
                p = (Lisp_Object *)(tr + next);
                next −= CELL;
                break;
            }
            break;
        }
    }
}
```