Distributed Garbage Collection

J. Dana Eckart
Richard J. LeBlanc

Georgia Institute of Technology

ABSTRACT

There are two basic approachs to the problem of storage reclamation, process- and processor-based, named for the view point used to recognize when a particular piece of storage can be reclaimed. Examples of the processor approach include mark/sweep and copying algorithms and their variants, while reference counting schemes use a process view of the collection. It is argued that the process approach is better suited for distributed computation where links between dynamically allocated objects may cross processor boundaries. In addition, the process approach allows the heap to be more conveniently shared with other processes in those cases when different processes might not have their own virtual address spaces. A new algorithm using the process approach is given. Its space requirement per object is better than that for reference counting. In addition, a restricted form of pointer replacement is supported which allows circular structures so constructed to be properly collected.

1. *Introduction*

Garbage collection, or more generally storage reclamation, is a common component of many interpreters. There are two basic approaches to the problem of garbage collection, which will be referred to as process- and processor-based collection. The distinction between these two approaches is important in the design of an algorithm for collecting structures which have links across machine boundaries in a distributed system. A distributed system (for purposes of this discussion) consists of a collection of machines, each with its own local memory, connected by a network such that pointers on one machine may directly address memory locations on another machine. Although related work in parallel or "on-the-fly" garbage collection[1] [2] [3] [4] [5] uses multiple processors, it differs due to the assumption that a single memory can be directly operated on by all of the processors.

Distributed garbage collection has been examined by both Bishop[6] and Hudak.[7] For Bishop the purpose was to divide data among separate memory areas[1] based upon its expected longevity. Areas with longer lived objects would be collected less frequently, thus reducing overhead due to garbage collection. Hudak's desire for investigating the problem is that distributed systems can take advantage of the natural parallelism found in functional languages. In this way, the inefficiencies, associated with garbage collection, of an applicative language can be overcome. The algorithm presented below was designed with the latter motivation. In either case, if the computations and results can be distributed over various machines in a system; a collection algorithm must take this distribution into account.

Bishop's algorithm requires each machine, X, to keep track of the objects (and the machines on which they exist) which have active pointers into X's address space. Normally, Bishop assumed, the number of such pointers (organized into

1. Each memory area can be viewed as belonging to a separate machine in the distributed system.

interarea link tables) would be small. However, in the event that a large number of pointers from a particular machine, Y, were noted, then pointers in the interarea link tables of machine X emanating from machine Y could be removed and replaced with a cable from machine Y. The cable is a more compact representation and is used when the number of interarea links is so great that it is just as well to do garbage collection on the two machines together (otherwise the pointers in the interarea link tables are used to indicate live data on machine X without the need for consulting other machines in the system during a garbage collection). Thus if machine X has a cable from Y then Y's address space will be collected whenever the address space of X is collected. The collection on each individual machine might be a copying (or mark/sweep) algorithm, in which case this would mean that both X and Y should finish copying (or marking) all live data before a flip (or sweep) is performed.

Hudak's algorithm makes use of what he calls a marking tree. Since the results of a computation for a functional language form a tree, the mark process of a mark/sweep algorithm is spawned on the root of the tree and is recursively spawned on each subtree. When marking reaches the leaves of the tree, they are coloured and then the direct ancestors are recursively coloured until the root is coloured. When the root of the tree is coloured, then the sweep phase can safely begin. Hudak's algorithm actually generalizes to directed graphs. This algorithm is useful because the colouring of the root of the result tree determines when the sweep phase can begin. Unfortunately, this approach requires that there only be one originator of computation on the distributed system, otherwise the machines in the system would have to poll each other in order to determine when all of the results (of the independent processes) had been completely marked.

The above two techniques are processor-based, since each processor collects structures based on processor boundaries. For example, in the mark/sweep algorithm all of the live data on a machine must be marked before sweeping begins otherwise some live data might be collected. In fact, algorithms which are typically called garbage collectors generally fall into this category. Process-based algorithms are those which do not require any knowledge of the state of the machine as a whole (e.g. has marking been completed?) in order to collect unused storage. Reference counting algorithms do exactly that.

Reference counting does not require a global action, since each structure contains enough information to determine when it is no longer being used. An object in the system gets this information from the process which created and used it. For example, in Lisp, the *car*, *cdr* and *cons* operators would pass information to objects on which they were invoked.

We feel that process-based collectors are better suited for applications on distributed systems. Since the life expectancy of objects is difficult to judge, it is likely that some machines in the network will have more objects of a particular life expectance than others. In such a situation, processor-based collectors like Bishop's algorithm could adversely affect other computations in the network due to one machine's need for garbage collection. The process approach does not suffer from this problem.

2. *The Reference Marking Algorithm*

Reference counting is well suited for distributed garbage collection because data structures can be collected independently from other computations. Because each object must keep a count of the number of references to it, reference counting requires *log* N additional bits for each object, where N is the total number of objects which can exist at one time. Like reference counting, reference marking is also a process-based collector but the space needed by the collector is bounded, three bits for each object and one additional bit for each pointer in the object.

The algorithm presented below is based on an idea proposed by Spector[8] in which pointers to structures are divided into two classes: defining and borrowed. A defining reference is the first, and often the only, reference to an object. If a copy of the pointer is made then the copy is a

borrowed reference since a portion of the data structure is being borrowed. While Spector states that such an algorithm would be well suited for garbage collection in "... systems which keep dictionaries of defined objects or in languages which control the way new user objects are defined.", he does not provide an algorithm. A particular difficulty in designing such an algorithm is that special care must be taken when values are returned from function calls in a list processing language such as *Lisp*. Consider the following *Lisp* example:

```
(def foo (x) (cons 'z (cdr x))
(foo '(a b c))
```

foo will return the list *(z b c)*, where the tail *(b c)* is a borrowed structure. The tail is borrowed since *(cdr x)* will return a copy of the pointer which already points to *(b c)*. When *foo* returns, some or all of the argument, *(a b c)*, which was used must be collected. Yet if this is done without regard for the value returned by *foo* a dangling reference will be left. What is needed is a mechanism for changing the borrowed reference into a defining one, as well as allowing the proper collection of the head of *(a b c)*.

2.1 The Algorithm

For convenience, we suppose that the algorithm is being used for a *Lisp*-like functional language, *Lisp**. The major differences between *Lisp** and *Lisp* are that it requires all referenced variables to appear as formal parameters in function declarations and no pointer replacement is allowed.

As was discussed above, the primary problem is recognizing when a borrowed reference should become a defining one, and to not continue collection beyond that point in the data structure when such a change is made. This necessitates recognizing shared objects. To do this two mark colours are used: red and green. By assuming that all of the arguments to a function call are the same colour, by marking the result value with a different colour and then collecting arguments with respect to the function colour, the shared objects can be identified. All arguments to a function can be made the same colour by alternating the colour assigned to function calls. Any colour assignment may be used for the initial function invocation; red will be chosen as the default.

Besides colour, objects have two additional features each of which may be encoded into a single bit. They are *shared-object* and *original-colour*. The first is a boolean value which indentifies shared objects, and the second records the original colour with which an object was coloured.

Two optimizations are used in order to increase the efficiency of the algorithm. First, if all of the arguments to a function call are borrowed references then it is unnecessary to mark the result since nothing will be collected. This optimization is useful for functions like *append* where recursive calls usually pass only borrowed references as parameters. The second optimization takes advantage of the way in which the primitive functions work. Consider *car*: if the argument to *car* is a defining reference, then the first cell of the structure can be returned to free storage without the need for any marking (likewise for *cdr*), otherwise nothing is collected. The remaining primitive functions have similar actions based on each function's semantics.

The above optimizations also require that the colour assigned to function call should only be alternated when a user defined (i.e. non-primitive) function is invoked. Since the result will be coloured opposite to the function colour, this makes the result the same colour as the parent function. In addition, since marking of the result is not done when all arguments are borrowed references, the function colour of such user defined functions is changed to the opposing colour before the body is evaluated. This ensures that the result will be of the appropriate colour.

The appendix contains pseudo-code which describes the reference marking algorithm. In order to more clearly understand reference marking, we present the following detailed example:

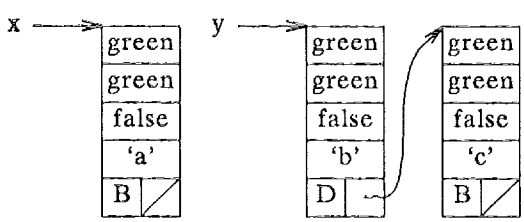
```
(def append (x y)
  (cond (x (cons (car x)
                 (append (cdr x) y)))
        (t y)))

(def skip2 (x) (cdr (cdr x)))

(skip2 (append (list a) (list b c)))
```

To further aid in describing the algorithm, the cons cells generated and the information they contain is presented as the steps are given. Starting from the top, working down line by line, the information encoded is: the *current-colour* of the cell; the *shared-object* flag; the *original-colour* of the cell; the car (with the reference kind[2] appearing to the left if a pointer to a cell); and finally the cdr (again with the reference kind to the left when needed).

1. *skip2* is invoked and assigned the colour red.
2. *append* is invoked and assigned the colour green.
3. The two arguments to *append* are evaluated. *x* and *y* are constructed by the primitive function *list* with their initial colouring being the colour of *list*, green. The arguments *x* and *y* of *append* are defining references.



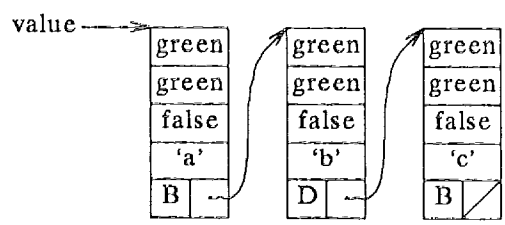
3.1 The body of *append* is begun. *cond* finds that a copy of the argument *x* is non-*nil* and thus begins evaluation of the corresponding function, *cons*, assigning it the colour green.

3.1.1 *cons* in turn needs to evaluate *(car x)* and *(append (cdr x) y)* as its arguments. The first is evaluated and returns the atom *a*.

3.1.2 A recursive call to *append* is made requiring that *(cdr x)* be evaluated to give the first parameter and a copy of *y* made for the other. This invocation of *append* is coloured red.

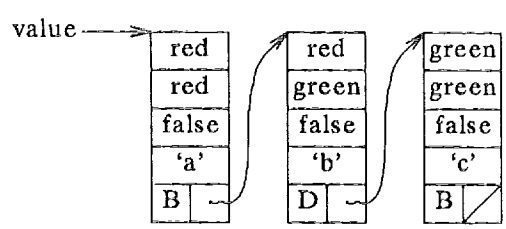
3.1.3 After the arguments for this invocation of *append* have been evaluated and both found to be borrowed references, the function colour is changed to green. Since *x* has the value *nil* the first condition of *cond* is false and the second action is taken, returns *y*. No marking is necessary since both arguments were borrowed references.

3.2 *cons* is now performed yielding the result:



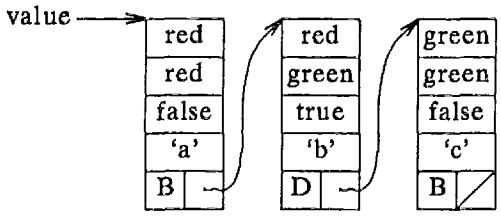
Note that the constructed object took its colour from the function colour of *cons*.

3.3 In turn, this is also the returned value of *append*. Since there is a parameter with a defining reference, *finish_function* will first *mark(value, red)* giving:



The first element of the list has both *current-colour* and *original-colour* coloured red, but the second element, refered to by the borrowed reference, only has the *current-colour* being red. *finish_function* will then *collect(x, green)* and *collect(y,green)* yielding:

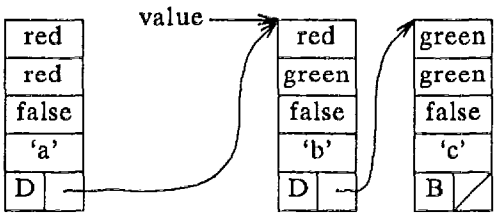
2. B is used to denote a borrowed reference while D refers to a defining reference.



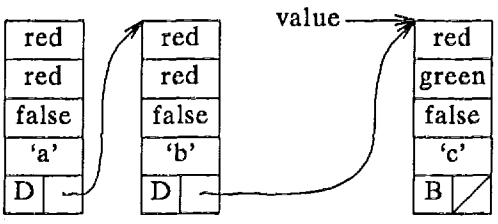
Note in particular that the head of the list is a copy, and that the original list *(a)* has been collected.

4. *skip2* now possess a defining reference to the list *(a b c)* and the evaluation of its body is begun.

5. Each *cdr* is performed in turn, returning as its value the *borrow_cdr* of its argument. After the first *cdr*, the value is:

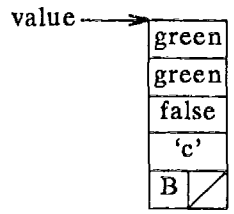


Note that the borrowed reference, contained in the first object, has become a defining reference and that the second object in the list is no longer a *shared-object*. The second *cdr* then gives:



Here, *propigate_colour_change* has propigated the *current-colour* to the last object while at the same time assigning the *original-colour* of the second object the value of its *current-colour*.

6. *finish_function* is called upon exit from *skip2*. Since the parameter was a defining reference, *mark(value, green)* and *collect(x, red)* are performed. Upon completion, *finish_function* returns a defining reference to:



As a side note, the final result is collected, after being displayed, by examining the tag of its first cell and using that for the call to *collect*. The initial evaluation request, *init* is then collected by *collect(init, red)*.

3. *Correctness of Reference Marking*

To show that reference marking behaves correctly it is necessary to show that it eventually collects all dead data and that live data is never collected. Data is live if it is reachable from some pointer on the execution stack; otherwise it is dead. The execution of a program is viewed as a tree of function calls, where the arguments to the call, represented by the siblings, are evaluated in parallel. Internal nodes of the tree indicate arbitrary function calls while leaves correspond to calls to primitive functions.

Claim 1: A borrowed reference to a *shared-object* becomes a defining reference. At the same time the referenced object also has *shared-object* set to false. The two operations form an atomic action.

Claim 2: Arguments to functions which are defining references, are completely collectable, with respect to that function's colouring, up to borrowed references of unshared objects.

Claim 3: Upon return, each function invocation collects all of the data which would be dead after return. No live data will be collected.

In order to see the first claim, remember that *mark* colours objects which are pointed to by borrowed references and ceases to colour further along that path in the data structure. Thus, when *collect* is freeing data and encounters an object of a non-collectable colour, that object is marked as shared. Since *collect* is only performed on

parameters to the function under consideration, it must be the case that this object marks the beginning of a shared structure which is being used as part of the result which the function is returning. Borrowed references to shared objects are not replaced by defining references immediately (and the object again becomes unshared). For efficiency reasons this change occurs during either a subsequent call to *collect*, *mark* or when the pointer is being borrowed as in the primitive functions *car* and *cdr*.

Remember that since *mark* colours objects pointed to by borrowed references, it is important to insure that this doesn't cause a 'break' in the colouring of a data structure that would cause *collect* to not finish collecting the entire structure. The second claim is that this will never happen. There are two actions in the algorithm which are relevant: (1) when a borrowed reference to a *shared-object* is made into a defining reference, remember that the *original-colour* colour is opposite of the *current-colour*. This informs *collect* that collection should continue but with respect to the *original-colour*; (2) when a collection on a data structure is performed and it collects up to a borrowed reference, the *current-colour* of the object to which it points is changed back to its *original-colour*. The first action allows the uppermost portion of a data structure to be marked with a different colour, but instructions are left so that the lower portion of the structure wiil be collected with the appropriate (opposite) colour. The second prevents results which have been marked but later dicarded which never caused an object to be marked as *shared-object* from having an opposing colour in the middle of a data structure. If this colouring remained then it might prevent the collection of the lower portion of the data structure.

To prove the third claim, *mark* and *collect* are examined in more detail. Since the result of a function may borrow from its arguments (represented in the tree by the results of the children of the current function node), it is necessary to *mark* the result first so that any *shared-object* which is also a part of one of the arguments will be coloured with the colour opposite to the one which will be used by *collect*. The colours will be different since *mark* and *collect* use opposite colours for their respective tasks. Those portions of the arguments which appear prior to an object marked with a different colour and before borrowed references are dead data (since the only way in which it would still be live would be for it to be passed back as a part of the result and this is exactly what the colouring of the result is used to detect). All of the dead data is therfore collected. To see that the result now considers that protion of the argument which was borrowed to be defining, remember that the *collect* action marked the object of different colour as a *shared-object* and by claim two above, the borrowed references will eventually become a defining reference.

By induction on the level of function call nesting (i.e. height of the execution tree) and claim three above, reference marking collects all and only inactive memory. The base case, that of a tree with one interior node, is true by an application of claim three. The inductive hypothesis is that if claim three is true for all children of an interior node then it is true for that node. By the inductive hypothesis, all of the children have collected their dead data and have not collected any live data. By arguments similar to those used in showing claim three, the parent node also properly collects dead data and returns no live data to free storage.

4. *Extending and Enhancing the Algorithm*

While reference marking is, in general, not capable of properly collecting arbitrary circular structures, if pointer replacement is restricted to cells that are reachable from a defining argument of the most recent non-primitive function invocation, then pointer replacement can be supported. If a pointer is replaced, then the replaced pointer must be remembered until the user defined function in which the replacement occurred is exited. At that time, the structure to which it points should be collected as if it were an argument to the function. This allows the proper collection of the pointer which was replaced. To properly collect the resulting, possibley circular, structure it is necessary

to add an additional colour so that no attempt is made to free cells which have already been returned to free storage. Unfortunately this also requires another bit per object to record the *current-colour*.

Allowing reassignment in a language can make it difficult to recognize parts of a computation which may be performed in parallel. However, by restricting pointer reassignment in the above way, the effect of a reassignment is localized. Thus there are fewer problems in recognizing when parallel computation is safe.

5. *Simulations and Results*

Each processor, in the simulation, is a stack machine which directly runs *Lisp**. The machines in the network are constructed so that the simulation occurs with respect to a global logical clock. Each machine is allowed to perform its computation for a short time, with all intermachine communications being queued by the receiving machine. A machine is chosen to be run next based upon its local clock. The machine with the minimum clock value is always run next with the minimum being recalculated each time a machine returns control to the main loop. Both idle and waiting times are also recorded for each machine. Idle time is accrued when there are no requests to be evaluated (either for the user or for collection purposes). Waiting time is the amount of time that a machine must wait for memory to be deallocated in order to satisfy the allocation request of some evaluation. All of the collection algorithms were implemented using stack traversal of data structures.

Bishop's algorithm has been implemented using a mark/sweep algorithm on each machine in the distributed system. Only cables have been assumed to exist between machines since the interconnectivety of the network was assumed to be high. Thus no inter-area link tables are used. The mark phase of the mark/sweep algorithm must occur as an atomic action but the sweep phase is incremental and it is indeed possible to begin another mark phase on a machine before the sweep phase has been completed (in which case the sweep phase is terminated). Multiprocessing on each processor is not simulated, thus when a process is blocked that machine may only service data retrieval requests for other machines in the network.

Reference marking, reference counting and Bishop's algorithm using mark/sweep were simulated on a single processor. For test programs designed to create large amounts of short lived data and varying amounts of long lived data, all of the algorithms required less than 6% of the total computation time. As expected, Bishop's algorithm (which is a simple mark/sweep on one processor) improved its performance dramatically with larger memory sizes. Both reference counting and marking required the same amount of time regardless of memory size, with reference counting using approximately 3.0 - 3.5% and reference marking only 2.4 - 2.7% of the total compuatation time.

While results are not yet available for multiple machine networks, tests have been designed to observe a behavior which should be unique to Bishop's algorithm. The test involves two machines, X and Y, with a cable from X into Y (i.e. X has pointers into the address space of Y). Almost all of Y's address space is garbage while X's is almost all live data. When Y needs to do a collection, the live data on both machines must be completely marked before a sweep phase can begin. However, Y must wait longer than it ordinarily might since much more marking will be required for X than for Y. Worse still, the data may not even belong to the same process. Thus those processes which are using machine X might be unfairly penalized by actions taken on machine Y since a garbage collection might not have been necessary from their prespective. Likewise, processes on machine Y must wait for the live data on X to be marked.[9]

6. *Conclusion*

We have argued that process-based rather than processor-based garbage collection techniques are better suited for collecting unused storage on distributed systems. The reference marking algorithm is an improvement over reference counting (another process-based collector) since the

space required to store the collection information is independent of the number of objects which can exist in the system. In addition, simulations indicate that reference marking is comparible in speed with other algorithms, especially when memory space is at a premium.

REFERENCES

1. Guy L. Steele Jr. (Sept 1975), Multiprocessing Compactifying Garbage Collection, *Communications of the ACM*, 18(9), pp 495 - 508.
2. Edsger W. Dijkstra and Leslie Lamport et al (Nov 1978), On-the-Fly Garbage Collection: An Exercise in Cooperation, *Communications of the ACM*, 21(11), pp 966 - 975.
3. Jeffrey L. Dawson (1982), Improved Effectiveness from a Real Time Lisp Garbage Collector, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp 159 - 167.
4. Mordechai Ben-Ari (July 1984), Algorithms for On-the-fly Garbage Collection, *ACM Transactions on Programming Languages and Systems*, 6(3), pp 333 - 344.
5. Ashwin Ram and Janak H. Patel (June 1985), Parallel Garbage Collection Without Synchronization Overhead, *12th Annual Symposium on Computer Architecture*, pp 84 - 90.
6. Peter B. Bishop (May 1977), Computer Systems With a Very Large Address Space and Garbage Collection, Massachusetts Institute of Technology, PhD Dissertation, TR-178.
7. Paul Hudak and Robert M. Keller (1982), Garbage Collection and Task Deletion in Distributed Applicative Processing Systems, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp 168 - 178.
8. David Spector (March 82), Minimal Overhead Garbage Collection of Complex List Structure, *ACM SIGPLAN Notices*, 17(3), pp 80 - 82.
9. J. Dana Eckart (Aug 1987), Garbage Collection in Distributed Systems, Ph.D. Dissertation, Georgia Institute of Technology.

Appendix

```
type pointer = record case ptr_type : (atom, cell) is
      atom:
         name : access string;
      cell:
         kind : (borrow, define);
         ptr  : access object;
   end record;

type Colour = (red, green);

type object = record
      original_colour, current_colour : Colour;
      shared_object : boolean;
      car, cdr : pointer;
   end record;

function borrow_car(P : pointer) : pointer is return borrow(prop_car(P)); end borrow_car;

function borrow_cdr(P : pointer) : pointer is return borrow(prop_cdr(P)); end borrow_car;
```

```
function borrow(P : pointer) : pointer is
   if P.ptr.ptr_type = cell then return pointer'(cell, borrow, P.ptr);
   else return P;
   end if;
end borrow;

function opposite_colour(C : colour) : colour is
   if C = red then return green; else return red; end if;
end opposite_colour;

function prop_car(P : pointer) : pointer is
   if P.ptr_type = atom or if P = NIL then return P; end if;
   elsif P.ptr.current_colour <> P.ptr.original_colour then
      propigate_colour_change(P);
      return P.ptr.car;
   elsif P.ptr.car.ptr_type = cell and if P.ptr.car.kind = borrow
        and if P.ptr.car <> NIL then
      if P.ptr.car.ptr.shared_object then
         P.ptr.car.ptr.shared_object := false;
         P.ptr.car.kind := define;
      end if;
      return P.ptr.car;
   else return P.ptr.car;
   end if;
end prop_car;

function prop_cdr(P : pointer) : pointer is
   if (P.kind = atom or if P = NIL) return P;
   elsif P.ptr.original_colour <> P.ptr.current_colour then
      propigate_colour_change(P);
      return P.ptr.cdr;
   elsif P.ptr.cdr.ptr_type = cell and if P.ptr.cdr.kind = borrow
        and if P.ptr.cdr <> NIL then
      if P.ptr.cdr.ptr.shared_object then
         P.ptr.cdr.ptr.shared_object := false;
         P.ptr.cdr.kind := define;
      end if;
      return P.ptr.cdr;
   else return P.ptr.cdr;
   end if;
end prop_cdr;

procedure propigate_colour_change(P : pointer) is
   if P.ptr.car.ptr_type = cell and if P.ptr.car <> NIL then
      if P.ptr.car.kind = define or P.ptr.car.ptr.shared_object then
          P.ptr.car.ptr.current_colour := P.ptr.original_colour;
      end if;
   end if;
   if P.ptr.cdr.ptr_type = cell and P.ptr.cdr <> NIL then
      if P.ptr.cdr.kind = define or P.ptr.cdr.ptr.shared_object then
          P.ptr.cdr.ptr.current_colour := P.ptr.current_colour;
      end if;
   end if;
   P.ptr.original_colour := P.ptr.current_colour;
end propigate_colour_change;
```

```
procedure collect(P : in out pointer; C : colour) is
   if P.ptr_type = atom or if P.ptr = NIL or if P.ptr.current_colour <> C
     or if (Not P.ptr.shared_object and P.ptr.kind = borrow) then return;
   elsif P.ptr.shared_object and P.kind = borrow then
      P.kind := define;
      P.ptr.current_colour := opposite_colour(P.ptr.current_colour);
      collect(P, opposite_colour(C));
   elsif P.kind = borrow then
      if P.ptr.current_colour <> P.ptr.original_colour then
         P.ptr.current_colour := P.ptr.original_colour;
      end if;
      return;
   elsif P.ptr.current_colour <> P.ptr.original_colour then collect(P, P.ptr.original_colour);
   elsif P.ptr.current_colour <> C then
      P.ptr.shared_object := true;
      return;
   end if;
   if P.ptr.car.ptr_type = cell and if P.ptr.car.ptr <> NIL then collect(P.ptr.car, C); end if;
   if P.ptr.cdr.ptr_type = cell and if P.ptr.cdr.ptr <> NIL then collect(P.ptr.cdr, C); end if;
   return_cell(ptr);
end collect;

procedure mark(P : in out pointer; C : Colour) is
   if P.ptr_type = atom or if P.ptr = NIL or if
     P.ptr.current_colour = C then
      return;
   elsif P.kind = borrow then
      if P.ptr.current_colour <> P.ptr.original_colour then propigate_colour_change(P); end if;
      if P.ptr.shared_object then P.kind := define; elsif return; end if;
      P.ptr.current_colour := C;
   else P.ptr.current_colour := C;
   end if;
   if P.ptr.car.ptr_type = cell and P.ptr.car.ptr <> NIL then mark(P.ptr.car, C); end if;
   if P.ptr.cdr.ptr_type = cell and P.ptr.cdr.ptr <> NIL then mark(P.ptr.cdr, C); end if;
end mark;

function finish_function (Value : in out pointer;
                          Args : argument_list; Fun_colour : Colour) : pointer is
   mark(Value, opposite_colour(Fun_colour));
   for arg in Args do collect(arg, Fun_colour)); end for;
   if Value.ptr_type = cell and if Value <> NIL
     and if value.kind = borrow and if Value.ptr.shared_object then
      Value.kind := define;
      Value.ptr.shared_object := false;
   end if;
   return Value;
end finish_function;
```