Language Implementor's Guide to the Garbage Collection Toolkit (draft)

Object Systems Laboratory

April 11, 1995

Contents

1 Introduction

This Guide describes the garbage collection toolkit developed at the Object Systems Laboratory. It concentrates on the language interface to the garbage collector, and does not describe the details of the garbage collection algorithm. It should be read in conjunction with other reports which provide the motivation for and evaluation of the toolkit approach and the collector implementation ([HMDW91],[HMS92]).

2 Packaging

2.1 Code provided

The toolkit source files are the following: {`blocks, cards, gc, gens, image, los, nurseries, pages, plan, protect, remsets, splay, ssb, steps`}.{`h, c`}, `boolean.h, convert.h, lang.h, minmax.h`.

These files, compressed and tarred, are available by anonymous ftp as file /pub/osl/packages/gc-0.4.tar.gz on `ftp.cs.umass.edu`.

2.2 Code files to be written by the implementor

It is convenient to place the procedures comprising the interface (see Figure ??) in a file `gc_lang.c`.

2.3 Auxiliary files to be written by the implementor

The plan file, which describes the memory outline for a particular language (or even a particular application).

3 Language — garbage collector interface

(Here we describe in broad terms the assumptions of the interface.)

3.1 The toolkit concept

For a detailed description, see [HMDW91].

3.1.1 Language runtime components

We distinguish three components of a running program for purposes of garbage collection, draw strict boundaries between them and define the interfaces. The first component is the garbage collector, charged with object and free space management. The second component is the language-specific part, charged with providing the garbage collector with the information it needs, such as the interpretation of objects. The third component is a set of 'remembering' techniques which the language part may use to manage this information.

```
\begin{figure}[htb]
%% Figure showing collector, language and remembering with call arrows
\centerline{\psfig{figure=structure.ps,width=5in}}
\caption{Toolkit structure.
Garbage collection protocol: 1) the language side invokes
collection; 2) the collector up-calls for object processing;
3) the language passes pointers to the remembering library to
be stored there and subsequently calls for them to be retrieved;
4) the library up-calls for processing of these pointers.}
\label{fig:structure}
\end{figure}
```

3.1.2 The garbage collection protocol

The language part may initiate garbage collection at any time (arrow 1, Figure **??**).

The collection may be a scavenge (`scavenge`), collecting several youngest generations, or a full garbage collection (`garbage_collect`).

The garbage collector responds by executing the collection algorithm. It calls back into the language (arrow 2), for initialization (`stop_threads`), for object processing[1] (`process_roots, process_scan, process_LO`), for finalization[2] (`clean_up_language`) and for language-specific policies.

The language makes use of the pointer-remembering techniques library, by calling it (arrow 3) to record entries in (for example) remembered sets (`add_remset_entry`), and later to retrieve them; the library calls back into the language (arrow 4) for processing of retrieved entries (`process_remset_entry`).

3.2 Object format

Objects are word-aligned, where a word is 32 bits. The toolkit makes no other assumptions about object format; the interpretation of object contents is entirely up to the language, when it *processes* the object.

[1]To *process* an object means to interpret the contents of the object, find all objects immediately reachable from it, and move them into to-space. When an object is moved, it is replaced in the from-space by a *forwarder object*. Again, it is the language side that is responsible for recognizing a forwarder.

[2]For example, finalization in our Smalltalk system involves adjusting the pointers in the method cache and in the virtual machine level debugger.

3.3 Large objects

Large objects are stored in separate areas and managed by threading and rethreading onto linked lists, rather than by copying [HMDW91]. The implementor will choose the threshold for maximum size of small objects; a suggested value is about 10% of the block size.

3.4 Mature objects

At present, the eldest generation must be tenured. In future releases of the toolkit, mature objects which become garbage will be collected using the train algorithm [?].

3.5 Routines provided by the collector

The following procedures are defined in the garbage collector and available to the language:

- The allocation macros are defined in `steps.h` (and normally imported by including `gc.h` in the language interface – in Smalltalk, they are imported into `alloc.c` and used for allocating Smalltalk-specific kinds of objects):
  - `ALLOCATE_NEW (void * return, int step_num, int longs):` Allocates an object of `long` longwords (slots) in the given step `step_num`. The address of the object is placed in `return`. If the step can't accommodate the object, and the step's *growth policy* returns *true*, the step is grown (by one block). If the object is too big to store even after that (i.e., it's larger than a block), -1 is returned for the address.
  - `ALLOCATE (void * return, step_info * step, int longs):` As above, but doesn't consult the step's *growth policy*.
  - `ALLOC_COPY (void * return, void * old, int longs):` Given the address of the old object, this macro allocates the space in the old object's default promotion step.
- In `blocks.h`:
  - `core_left:` Provides an indication of the amount of heap space remaining for allocation (in bytes).
  - `call_at_bytes_left:` Signal when available heap space is low.
- In `gc.h`:
  - `init_gc:` Initializes generation, block and step data structures of the collector. It must be called exactly once, before any allocation.
  - `scavenge (int n):` Collects generations `n` and younger. It up-calls the language interface procedures `stop_threads` and `process_roots`; it then calls `basic_scan`, passing it the scanning procedures `process_scan` and `process_LO` of the language interface. Procedure `basic_scan` performs the Cheney breadth-first copy, applying the scanning procedures passed to scan individual blocks and large objects.

- `garbage_collect`: Collects all generations.
- `scan`: Prepares all steps for scanning and passes scanning procedures to `basic_scan`. It is not necessary for garbage collection to call this function explicitly, but it may be useful for examining heap contents. However, this function is called implicitly from within garbage collection when debugging is enabled, and then it requires the existence of language-supplied functions check_scan and check_LO.

- In `nurseries.h`:
  - ...

- ...

3.6 Interface routines needed by the collector

The implementor must write these procedures, which the garbage collector will call (see the Smalltalk example):

- `stop_threads`: The language stops the running mutator threads so scavenge can proceed.
- `process_roots`: The language processes the root pointers of the scavenge, moving a number of objects into to-space.
- *Scanning procedures:*
  - `process_scan`: The language processes the last batch of objects that have been moved into to-space, causing a new batch to be moved there.
  - `process_LO`: The language processes a large object that has been logically moved into to-space, causing a new batch of objects to be moved there.
  - `check_scan, check_LO`: The language processes the objects, checking pointer consistency – for debugging purposes only.
- `clean_up_language`: The language updates any data structures (globals, registers, etc.) that may point to objects moved. This procedure will be invoked by the garbage collector following each scavenge.
- *A set of policy procedures for generations and steps:* Policy routines return policy decisions. At present, the decisions are whether to grow a step, and whether to scavenge a step (or a generation). A number of standard and default policy routines are already available in `policy.c`.[3]

3.7 Routines provided by the remembering library

The set of routines available depends on the remembering technique chosen.

[3] A policy that always returns *true*, a policy that always returns *false*, a step-growing policy that returns *true* when the step is smaller than its predeclared maximum, step-scavenging policy that returns *true* when the step size reaches this predeclared maximum.

3.7.1 Using 'remembered sets'

This technique maintains a hashed table, each entry of which records the (address of the) word ("slot") that was updated. (Variations: 1° 'slots' records the slot itself; 2° 'objects' records the (address of the) object only, requiring object to be scanned later.) The natural hash table slot is computed by bitwise operations on the address to be recorded. The entry can be placed either in this, or in one of the several succeeding hash table slots, whichever is free. If there is no room left for an entry, a new, larger hash table is allocated and the entire set is rehashed. For efficiency, the sets can also shrink as needed. Separate sets are kept for each generation; prior to scavenging, they are merged.

These are the procedures provided (see `remsets.c`):

- `alloc_remset`: Allocates initial space for a set; at present, this is a little over 256 entries. (Some space is also taken up by the remset header information – the current size, the number of slots in use.) All the entries are cleared.
- `add_remset_entry`: Inserts an entry (a slot address) into the set, hashing it into the appropriate place. If the remset becomes too full (namely, if it is filled above `REMSET_GROW_RATIO` = 60%), it is grown.
- `check_remset`: An auxiliary function which examines a remset and computes its (percent) collision rate as the fraction of entries out of their natural place. (Diagnostic use only.)
- `merge_remsets (source, target)`: Copies all entries from the source set into the target set. The source set is cleared. If the source set was originally less than one-third (that is, $\frac{1}{REMSET\_SHRINK\_DIVISOR}$) full, it is halved in size.
- `process_remset (remset, process)`: Applies a procedure 'process' to each non-null entry in the set.
- `process_remset_clear (remset, process)`: As above, but also removes processed entries from the set.
- `probe_remset`: Check if a particular entry is in the set. (Diagnostic use only.)
- `print_remset`: Displays the entries in a set (for diagnostic purposes only).

These must be written in the language interface:

- *procedures that process remembered set entries, to be passed as arguments to* `process_remset` (In Smalltalk, we have the procedure `process_remset_entry` that checks if the value stored is a pointer etc. – see below.)

3.7.2 Using card marking

This technique uses a table to mark cards (small, fixed-size, memory regions) that have been stored into. (Variations: 1° - 'slots': marks the card of the slot that was modified; 2° - 'objects':

marks the card of the object header for the object that contains the slot, which may be the same card as in variation 1° or a preceding card.)

The size of a card in bytes must be a power of 2; for 2^n specify n as the value of the `CARD_SHIFT` constant in `cards.h`.

These are the procedures provided:

- `init_cards`: Must be called from the language interface (exactly once, at the beginning) to initialize the card table.
- `process_cards`: Applies `process_card` (or `process_aligned_card`) to each dirty card of the generations not scavenged.
- `process_LO_cards`: Applies `process_LO_range` to all dirty cards corresponding to large objects in generations not scavenged.

These must be written in the language interface:

- `process_card`:
- `process_aligned_card`:
- `process_LO_range`:

3.7.3 Using page marking

This technique uses the system–provided page tables instead of cards.

These are provided:

- `init_pages`: Initializes the page table ??
- `process_pages`:
- `process_LO_pages`:

These must be written in the language interface:

- `process_page`:
- `process_aligned_page`:
- `process_LO_range`:

3.8 Interface routines needed by the remembering library

4 Configuration

We describe compile-time and run-time adjustable parameters.

4.1 Choice of remembering technique

The remembering library supports several remembering schemes; the implementor will choose one, and invoke the appropriate procedures.

4.2 Pointer filtering choices

The garbage collector provides support for generational filtering; it is up to the language to filter non-pointers etc., some of this work may be done at compile-time in statically-typed languages.

4.3 Allocation and promotion strategies: the plan file

In the original toolkit design, a *plan file* was envisioned. The plan file defines the number of generations, the number of steps in each, and the maximum number of blocks in each step. It also specifies, for each step, to which step to promote objects that survive the scavenge.

However, the plan file has not yet been integrated into the toolkit. At present, our Smalltalk and SML language interfaces provide their individual plan files. For example, the Smalltalk plan file should contain two lines for each generation and for each step as follows:

`gen=`*g* `first_step=`*s*
`step=`*s* `gen=`*g* `scannable=`*s* `prom_step=`*s* `max_blocks=`*n*
(See Smalltalk example.)

5 Example: Smalltalk

(Discuss frames, preallocation, loading the image.)[!]

We have implemented the interface part using several remembering techniques, namely, remsets, cards, and pages, for the purpose of comparing their efficacy . In the following, we only discuss using remsets for recording slots (the compile-time option `REM_SLOTS`).

For Smalltalk, the following procedures are used in the language interface part:

- `init_gc_lang`: Initializes remembered sets (or cards, or pages, as appropriate).
- `(private) remember_store (OOP * src_slot, int src_gen, OOP src_obj)`: This procedure is inlined and called at each assignment to a pointer slot. (In a compiled language, the compiler would have to emit extra code to do this.) Here, `src_slot` is the address of the slot where the pointer is to be stored, `src_gen` is the generation of the slot, and `src_obj` is the new pointer value stored. The procedure `remember_store` first makes sure that this value is not nil, and that, as a an object pointer, it is a valid one. It then goes on to compute the generation corresponding to the new pointer value (*target generation*). If this generation is younger than the generation of the slot itself (*generation filtering*), the store is recorded. The actual recording work is delegated to procedure `add_remset_entry` of the toolkit, which is passed the address of the remset for the target generation, and the slot address.

- `(private) MOVE_FILTER`: Computes if a Smalltalk object belongs to a generation being scavenged.

- `(private) copy_slots`: An auxiliary procedure that copies a given number of slots (i.e., words) from one location to another.

- `(private) move_object`: Used to move a Smalltalk object from its previous location in old-space to new-space. It needs to understand the Smalltalk object format. If the object at the old location has already been moved, it returns the forwarding address immediately. Otherwise, it it proceeds according to the kind of object. It determines the amount of space needed for the copy (from the Smalltalk object header information), allocates this amount in the to-region (using `ALLOC_COPY`[!]), and then copies the contents of the object using `copy_slots`. Finally, it modifies the header to reflect the fact that the object has been forwarded and mark the new address. A special case is for *large objects*, which require the procedure `copy_LO` to be invoked [!].

- `(private) update_slot`: This procedure is used when *scanning* a moved object during scavenging. A non-pointer slot is ignored; a pointer slot is ignored if it does not satisfy the `MOVE_FILTER`, that is, if it refers to an object that's not to be moved on this scavenge. Otherwise, the object referred to is moved into to-space using `move_object`, and the slot is updated accordingly. Also, the store is recorded using `remember_store`.

- `process`: Goes through the slots of a Smalltalk object, invoking `update_slot` on each in turn (and also on the class object). This procedure is made available to the collector.

- `process_scan`: This procedure is only used by the garbage collector, which passes it to `basic_scan`.

- `(private) process_remset_entry`: The entries in remembered sets are part of the root set for a scavenge. Given a remset 'entry' (actually, a slot address) this procedure determines the generation of the object the slot belongs to. The way to do this is to first see whether the slot is in a small object block or a large object block. In the former case, `GEN_NUM`, a shift macro, is used; in the latter case, the function `get_LO_gen` must be called. If the slot is in a generation that is itself scavenged this time, this entry may safely be ignored. Otherwise, if the entry refers to an object in from-space of this scavenge, it is moved. Lastly, the slot is recorded again using `remember_store` for the next scavenge.

- `(private) process_remsets`: This procedures dumps the remembered sets of all the generations being scavenged into one set, using `merge_remsets`, thus removing duplicate entries, and then processes and clears all the entries by passing `process_remset_entry` to `process_remset_clear`, a toolkit procedure.

- `process_roots`: Processes all the roots of a scavenge: the ones in the remembered sets, and the ones in Smalltalk global variables maintained by the interpreter (for example, the SmallInteger class, and the Smalltalk dictionary object). This procedure is made available to the collector.

- `(private) check_slot`, `(private) check`, `check_scan` are there for debugging purposes only, and check the consistency of the Smalltalk heap following scavenges. Similar procedures can be written for other languages, however. They are triggered from the garbage collector immediately before, and immediately after a scavenge (when the compile-time flag DEBUG is specified). Procedure `check_scan` is made available to the collector, just like `process_scan`.

- `(private) fix_method_cache`: The interpreter maintains a small cache with method addresses to avoid having to do a full method lookup (search a receiver's method dictionary (and possibly traverse the Smalltalk class hierarchy)) on each message send. This cache is implemented as a hash table; the class object and the message selector are used to index into it, and the bytecode method object is retrieved. Any of these might be moved during a scavenge, so `fix_method_scavenge` runs through this table and updates all the entries that have been invalidated.[!]

- `clean_up_language`: Calls `fix_method_cache` and `init_small_class_table`, which adjusts an array of global Smalltalk objects. This procedure is made available to the collector.

Block size. The block size we use is soft-coded in `blocks.h` and is 2^{16} bytes. Changing the block size requires modifying the constant `BLOCK_SHIFT`, but we find the present value satisfactory.

Card size. When using one of the card-marking schemes for store recording, the size of the cards must be specified using constant `CARD_SHIFT` in the file `cards.h`. A size of 256 or 512 appears to be a good choice for a range of Smalltalk programs we examined [HMS92].

Plan file. The plan file we use for Smalltalk is the following:

```
gen=0 first_step=0
step=0 gen=0 scannable=1 prom_step=0 max_blocks=10
step=1 gen=0 scannable=1 prom_step=2 max_blocks=1
gen=1 first_step=2
step=2 gen=1 scannable=1 prom_step=3 max_blocks=1
gen=2 first_step=3
step=3 gen=2 scannable=1 prom_step=4 max_blocks=1
gen=3 first_step=4
step=4 gen=3 scannable=1 prom_step=5 max_blocks=10
gen=4 first_step=5
step=5 gen=4 scannable=1 prom_step=5 max_blocks=80
```

Step 0 in generation 0 is reserved for allocating frames, and objects are never promoted out of it; step 1 is the *nursery* step. The succeeding generations have one step each. The step in generation 3 is allowed to grow up to 10 blocks in size, while the oldest step, in generation 4, which contains the *tenured* objects, is allowed to take 80 blocks. This is not a hard limit on total heap allocation, as explained under *policies*.

Policies. The *step policies* have to do with deciding when to increase the number of blocks in a step, and when to scavenge a step. The *generation policy* decides when to scavenge a generation. The default association of policies with steps and generations is hard-wired in the toolkit code: a step grows until its `max_blocks` limit is reached and is scavenged if it is exceeded, except that the tenured step is always grown and never scavenged. A generation is scavenged if any of its steps must be scavenged.

Some useful simple policies are given in `policy.c`. See `steps.c` and `gens.c` for details.

6 Example: SML

The toolkit was integrated with Standard ML of New Jersey in its version 0.75, and later upgraded to versions 0.91 and 0.93. The description below assumes some familiarity with the organisation of the SML/NJ compiler, which we left intact, and of the run-time system, large parts of which are used in our version.

SML has been said to allocate heap objects at "a furious rate". The large number of objects and the wide span of their sizes served to exercise the algorithms of the toolkit extensively; a good number of implementation oversights were discovered in the process.

Data revealed to the garbage collector in SML/NJ come in three varieties: 1) objects allocated in the heap by the ML code, 2) static data allocated by the runtime (in partiular, the C-thread image of the ML registers), and 3) precompiled ML code (text segment of the executable image). Only the first kind is subject to garbage collection; however, pointers may point outside the traditional object allocation area (arena in SML/NJ terminology) and still be valid. This circumstance was not too helpful when we wrote the `check_scan` code. Another consequence was the need to assign a generation number to the objects outside the heap; we use one plus the number of the oldest true generation.

6.1 Allocation

SML/NJ makes the assumption of a contiguous allocation region, bounded from below by the allocation pointer (kept in a register) and from above by a limit pointer (kept, suitably adjusted, in another register). At convenient points in the ML code, explicit checks are performed against these pointers, and control is passed to the collector if space may be insufficient. (See [A runtime system, by Appel] for details, but note that the collection triggering mechanism has undergone changes in more recent versions of SML/NJ.) The relevant inputs for the collector are the C data structure MLState, containing all ML-thread registers, a bit-pattern (mask), indicating which of the general-purpose registers are live (in the compiler sense of liveness), and the amount of space that needs to be made available before resuming the ML thread.

We map ML's allocation region to a nursery in step 0. (This is an upward-growing nursery, with no protection.) We found a nursery size of several megabytes to be satisfactory.

Observe that all objects, regardless of size, are allocated in this region. This has two consequences: 1) large objects surviving one scavenge must be copied from the nursery into the LOS, which according to our measurements is not too detrimental to performance; and 2) a runtime check is needed following each scavenge to see if the nursery is large enough, and if not, it must be reallocated.

6.2 Write barrier and root processing

The write barrier in SML/NJ is implemented with some assistance from the compiler. Wherever a location (a *ref* cell or an array element is stored into, a record is pushed onto a linked list of stored-into locations. This store-list is accessed via a dedicated ML register. This approach is feasible only because the frequency of updates in ML is low relative to the rate of allocation.

We decided to keep this mechanism intact, but had to augment it with remembered sets for our multi-generational setup. On each scavenge, we empty the store-list into a *process set*, and then add to it the entries from the remembered sets of generations to be collected. On promotion, these sets are regenerated. [4]

The remaining roots for the scavenge are in the registers marked live and a handful of global variables.

6.3 Object processing

We left the object format unchanged, and were able to use the SML/NJ procedure `xgc` as front end in analysing objects; promotion is somewhat complicated by the need to differentiate between small and large objects.

6.4 Major collections

In our implementation of Smalltalk, we rely on the interactive user to initiate major collections (collections of the entire heap). In SML this approach doesn't work, since the amounts allocated in between successive user interaction points may be large enough to warrant major collections. Our mechanism is to make a decision following each minor scavenge. We have implemeneted two policies. The first policy relies on exact data size calculations, and a conservative estimate of the live data size. Thus, this policy is the safest, but also the costliest to use both in per-scavenge computation time and in the frequency of major scavenges. The other policy estimates data size based on the number of blocks occupied, and is thus a safe approximation. It's much cheaper to compute and we use it whenever we are not gathering statistics.

6.5 Statistics

We developed our version of SML with a view to gathering object lifetime and other statistics by varying the many parameters the toolkit provides. Especially of interest are detailed step size printouts, and object size printouts.

7 Example: Modula-X

Modula-X is a small Pascal-like subset of Modula-3. We developed a toy compiler for this language in the broader context of explorations in compiling technology, which are irrelevant to this document. The compiler produces C code acceptable to the GNU C compiler, `gcc`. Therefore, it's an implementation model different from the ones described above. As usual, we had to deal with object allocation, with accurate tracing of roots from the stack into the heap, and the intra-generational pointers. We had to express these operations within the syntax of C, and in a way that would preserve semantic correctness under code manipulation by the C compiler.

For allocation, we use a contiguous region of memory called the nursery. Garbage collection is triggered when this region overflows. We use an explicit check for the purpose. (The alternative is to use memory protection mechanisms.) Large objects are immediately allocated

[4] Observe the use of the macro `NEW_GENERATION` and the toolkit function `get_LO_new_gen` in the context of filtering newly created pointers.

in a separate area and are not copied at collection time. (We could statically determine the size of objects and inline either of the allocation calls, but we forgot to do so in the present compiler.)

To start a collection, we must be able to find all the references in the stack. (See [DMH92] for a solution of this problem in a real implementation.) These references have been mapped to C pointers, and they are certainly at known offsets within the C stack frames. However, since we do not have a way of syntactically expressing the frames, in the presence of recursion we must incur a run-time cost, and keep a dedicated stack for collection root pointers. Therefore, we emit code at the beginning of each block to push the addresses of any locally declared pointers, and we emit code at the end of each block (and also before procedure returns and loop exits) to discard them. Note also that taking the addresses of these pointers and storing them keeps the C compiler from performing optimisations, such as pointer arithmetic or register allocation, that could compromise code correctness under garbage collection.

We must also keep track of intragenerational references. We use remembered sets to implement the write barrier. (See [HMS92] for a discussion and comparison of this and other approaches). There is one remembered set for old pointers into each generation and one, the process set, for newly recorded pointers. For each assignment to an l-value of a reference type, the compiler tries to establish whether it lies on the stack or in the heap, and in the latter case, emits code to update the process set. (Note that if the l-value is a parameter passed by reference, the compiler cannot in general know where it is; if it is on the stack, the collector interface then discards this pointer at run-time because its source address is seen not to be in object space.)

Let us illustrate the model with an example. The following Modula-X program repeatedly builds a tree, traverses it and replaces subtrees.

```
MODULE Main;

TYPE CARDINAL = INTEGER;
TYPE CHAR = INTEGER;
TYPE REFANY = INTEGER;


CONST height = 9;
CONST branching_factor = 4;
CONST bulk = 0;
CONST destroy_depth = 3;
CONST niters = 100;


TYPEREC HeapNode = RECORD
  height: INTEGER;
  bulk: REF ARRAY [1..bulk] OF REFANY;
  children: ARRAY [1..branching_factor] OF REF HeapNode;
END;
```

```
TYPE Node = REF HeapNode;



(*****************************************************************)
VAR last: INTEGER := 0;
PROCEDURE random_int (max: INTEGER): INTEGER =
BEGIN
  last := (last * 17 + 5) MOD (max);
  RETURN last + 1;
END random_int;
(*****************************************************************)

PROCEDURE construct (height: CARDINAL): Node =
VAR anode: Node;
    i: INTEGER := 1;
BEGIN
  anode := NEW (Node);
(*  anode^.bulk := NEW (REF ARRAY [1..bulk] OF CHAR); *)
  anode^.height := height;
  IF (height = 1) THEN RETURN anode; END;
  WHILE (i <= branching_factor) DO
    anode^.children[i] := construct (height - 1);
    i := i + 1;
  END;
  RETURN anode;
END construct;

(* Walk down STEPS random_int steps and destroy the tree at that height and
   replace it by a new one *)

PROCEDURE destroy (root: Node; steps: INTEGER;
                  branching_factor, bulk: CARDINAL) =
VAR current: Node;
    temp: INTEGER;
BEGIN
  current := root;
  WHILE (steps > 1) DO
    (* generate a random number between 0 and branching factor to decide
       which child to go to *)
    WRITE("**DESCEND ONE STEP\n");
    temp := 1; (*random_int (branching_factor); *)
    current := current^.children[temp];
```

```
    steps := steps - 1;
  END;
  temp := 1; (* random_int (branching_factor);*)
  WRITE("**REPLACING...\n");
  current^.children[temp] :=
    construct (current^.height - 1);
END destroy;

VAR root: Node;
    i: INTEGER := niters;
BEGIN
  root := NEW (Node);
  WHILE (i > 0) DO
    WRITE("**CONSTRUCT:%d\n",i);
    root :=  construct (height);
    WRITE("**DESTROY:%d\n", i);
    destroy (root, destroy_depth, branching_factor, bulk);
    i := i - 1;
  END;
END Main.
```

We use the following templates:

```
typedef struct __typedescriptor
{
  int __size;
  struct __objectheader * (*__scan_procedure)();
} __typedescriptor;


typedef struct __objectheader
{
  int __forwarded;
  struct __typedescriptor * __type_desc;
} __objectheader;


........
```

In this program, the type `HeapNode` is a type (the only) for which values are allocated on the heap. For each such type, the compiler emits a type descriptor and type-specific procedures `new` and `scan`.

```
__objectheader * __scan___type_2_ ( __objectheader * obj, int gen )
{char * slot;
slot = ( char * ) obj + 12 ;
if ( * (int * ) slot != 0) {update_scanned_slot (( __objectheader * * ) slot, gen);}
slot = ( char * ) obj + 16 ;
if ( * (int * ) slot != 0) {update_scanned_slot (( __objectheader * * ) slot, gen);}
slot = ( char * ) obj + 20 ;
if ( * (int * ) slot != 0) {update_scanned_slot (( __objectheader * * ) slot, gen);}
slot = ( char * ) obj + 24 ;
if ( * (int * ) slot != 0) {update_scanned_slot (( __objectheader * * ) slot, gen);}
slot = ( char * ) obj + 28 ;
if ( * (int * ) slot != 0) {update_scanned_slot (( __objectheader * * ) slot, gen);}
return ( ( __objectheader * ) ( ( char * ) obj + 32 ) );
}

__typedescriptor  __typedesc___type_2_ =
{ 24 , (void *) & ( __scan___type_2_) };

_type_2_ * __new___type_2_ ()
{
_type_2_ * x;
x = ( (_type_2_ * ) _myalloc (32));
((__objectheader * ) x)->__forwarded = 0;
((__objectheader * ) x)->__type_desc = & __typedesc___type_2_;
return x;
}
__objectheader * __scan___type_4_ ( __objectheader * obj, int gen )
{char * slot;
return ( ( __objectheader * ) ( ( char * ) obj + 8 ) );
}
```

Object scanning therefore simplifies to the following:

```
void process_scan (__objectheader * ptr, __objectheader * limit, int gen_num)
{
  __objectheader * newptr;
  __objectheader * (*appropriate_proc)();

  while (ptr < limit)
    {
      appropriate_proc = ptr->__type_desc->__scan_procedure;
      newptr = appropriate_proc (ptr, gen_num);
```

```
      ptr = newptr;
    }
}
```

Here is the complete translation into C:

```
....to be added.....
```

References

[DMH92] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992. SIGPLAN, ACM Press.

[HMDW91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992. *ACM SIGPLAN Notices 27*, 10 (October 1992).