9

Garbage Collection

The execution model of a program on a microprocessor corresponds to that of imperative programming. More precisely, a program is a series of instructions whose execution modifies the *memory state* of the machine. Memory consists mainly of values created and manipulated by the program. However, like any computer resource, available memory has a finite size; a program trying to use more memory than the system provides will be in an incoherent state. For this reason, it is necessary to reuse the space of values that are at a given moment no longer used by future computations during continued execution. Such memory management has a strong influence on program execution and its efficiency.

The action of reserving a block of memory for a certain use is called *allocation*. We distinguish *static allocation*, which happens at program load time, *i.e.* before execution starts, from *dynamic allocation*, which happens during program execution. Whereas statically allocated memory is never reclaimed during execution, dynamically allocated regions are susceptible to being freed, or to being reused during execution.

Explicit memory management is risky for two reasons:

- if a block of memory is freed while it contains a value still in use, this value may become corrupted before being accessed. References to such values are called *dangling pointers*;
- if the address of a memory block is no longer known to the program, then the corresponding block cannot be freed before the end of program execution. In such cases, we speak of a *memory leak*.

Explicit memory management by the programmer requires much care to avoid these two possibilities. This task becomes rather difficult if programs manipulate complicated data structures, and in particular if data structures share common regions of memory.

To free the programmer from this difficult exercise, automatic memory management mechanisms have been introduced into numerous programming languages. The main

idea is that at any moment during execution, the only dynamically allocated values potentially useful to the program are those whose addresses are known by the program, directly or indirectly. All values that can no longer be reached at that moment cannot be accessed in the future and thus their associated memory can be reclaimed. This deallocation can be effected either immediately when a value becomes unreachable, or later when the program requires more free space than is available.

Objective Caml uses a mechanism called *garbage collection* (GC) to perform automatic memory management. Memory is allocated at value construction (*i.e.,* when a constructor is applied) and it is freed implicitly. Most programs do not have to deal with the garbage collector directly, since it works transparently behind the scenes. However, garbage collection can have an effect on efficiency for allocation-intensive programs. In such cases, it is useful to control the GC parameters, or even to invoke the collector explicitly. Moreover, in order to interface Objective Caml with other languages (see chapter 12), it is necessary to understand what constraints the garbage collector imposes on data representations.

Chapter Overview

This chapter presents dynamic memory allocation strategies and garbage collection algorithms, in particular the one used by Objective Caml which is a combination of the presented algorithms. The first section provides background on different classes of memory and their characteristics. The second section describes memory allocation and compares implicit and explicit deallocation. The third section presents the major GC algorithms. The fourth section details Objective Caml's algorithm. The fifth section uses the `Gc` module to control the heap. The sixth section introduces the use of *weak pointers* from the `Weak` module to implement caches.

Program Memory

A machine code program is a sequence of instructions manipulating values in memory. Memory consists generally of the following elements:

- processor registers (for direct and fast access),
- the stack,
- a data segment (static allocation region),
- the heap (dynamic allocation region).

Only the stack and the dynamic allocation region can change in size during the execution of a program. Depending on the programming language used, some control over these classes of memory can be exercised. Whereas the program instructions (code) usually reside in static memory, dynamic linking (see page 241) makes use of dynamic memory.

Allocation and Deallocation of Memory

Most languages permit dynamic memory allocation, among them C, Pascal, Lisp, ML, SmallTalk, C++, Java, ADA.

Explicit Allocation

We distinguish two types of allocation:

- a simple allocation reserving a block of memory of a certain size without concern of its contents;
- an allocation combining the reservation of space with its initialization.

The first case is illustrated by the function `new` in Pascal or `malloc` in C. These return a pointer to a memory block (*i.e.* its address), through which the value stored in memory can be read or modified. The second case corresponds to the construction of values in Objective Caml, Lisp, or in object-oriented languages. Class instances in object-oriented languages are constructed by combining `new` with the invocation of a constructor for the class, which usually expects a number of parameters. In functional languages, constructor functions are called in places where a structural value (tuple, list, record, vector, or closure) is defined.

Let's examine an example of value construction in Objective Caml. The representation of values in memory is illustrated in Figure 9.1.

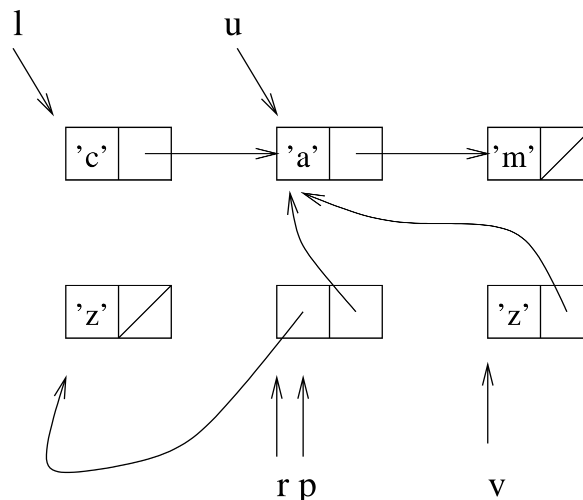

Figure 9.1: Memory representation of values.

```
# let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
val u : char list = ['a'; 'm']
```

```
# let v = let r = ( ['z'] , u )
          in match r with p → (fst p) @ (snd p) ;;
val v : char list = ['z'; 'a'; 'm']
```

A list element is represented by a tuple of two words, the first containing a character and the second containing a pointer to the next element of the list. The actual runtime representation differs slightly and is described in the chapter on interfacing with C (see page 315).

The first definition constructs a value named `l` by allocating a cell (constructor ::) for each element of the list `['c';'a';'m']`. The global declaration `u` corresponds to the tail of `l`. This establishes a sharing relationship between `l` and `u`, *i.e.* between the argument and the result of the function call to *`List.tl`*.

Only the declaration `u` is known after the evaluation of this first statement.

The second statement constructs a list with only one element, then a pair called `r` containing this list and the list `u`. This pair is pattern matched and renamed `p` by the matching. Next, the first element of `p` is concatenated with its second element, which creates a value `['z';'a';'m']` tied to the global identifier `v`. Notice that the result of `snd` (the list `['a';'m']`) is shared with its argument `p` whereas the result of `fst` (the character `'z'`) is copied.

In each case memory allocation is explicit, meaning that it is requested by the programmer (by a language command or instruction).

Note

Allocated memory stores information on the size of the object allocated in order to be able to free it later.

Explicit Reclamation

Languages with explicit memory reclamation possess a freeing operator (`free` in C or `dispose` in Pascal) that take the address (a pointer) of the region to deallocate. Using the information stored at allocation time, the program frees this region and may re-use it later.

Dynamic allocation is generally used to manipulate data structures that evolve, such as lists, trees *etc.*. Freeing the space occupied by such data is not done in one fell swoop, but instead requires a function to traverse the data. We call such functions destructors.

Although correctly defining destructors is not too difficult, their use is quite delicate. In fact, in order to free the space occupied by a structure, it is necessary to traverse the structure's pointers and apply the language's freeing operator. Leaving the responsibility of freeing memory to the programmer has the advantage that the latter is sure of the actions taken. However, incorrect use of these operators can cause an error during the execution of the program. The principal dangers of explicit memory reclamation are:

- dangling pointers: a memory region has been freed while there are still pointers pointing at it. If the region is reused, access to the region by these pointers risks being incoherent.
- Inaccessible memory regions (a memory "leak"): a memory region is still allocated, but no longer referenced by any pointer. There is no longer any possibility of freeing the region. There is a clear loss of memory.

The entire difficulty with explicit memory reclamation is that of knowing the lifetime of the set of values of a program.

Implicit Reclamation

Languages with implicit memory reclamation do not possess memory-freeing operators. It is not possible for the programmer to free an allocated value. Instead, an automatic reclamation mechanism is engaged when a value is no longer referenced, or at the time of an allocation failure, that is to say, when the heap is full.

An automatic memory reclamation algorithm is in some ways a global destructor. This characteristic makes its design and implementation more difficult than that of a destructor dedicated to a particular data structure. But, once this difficulty is overcome, the memory reclamation function obtained greatly enhances the safety of memory management. In particular, the risk of dangling pointers disappears.

Furthermore, an automatic memory reclamation mechanism may bring good properties to the heap:

- *compaction*: all the recovered memory belongs to a single block, thereby avoiding fragmentation of the heap, and allowing allocation of objects of the size of the free space on the heap;
- *localization*: the different parts of the same value are close to one another from the point of view of memory address, permitting them to remain in the same memory pages during use, and thereby avoiding their erasure from cache memory.

Design choices for a garbage collector must take certain criteria and constraints into account:

- reclamation factor: what percentage of unused memory is available?
- memory fragmentation: can one allocate a block the size of the free memory?
- the slowness of allocation and collection;
- what freedom do we have regarding the representation of values?

In practice, the safety criterion remains primordial, and garbage collectors find a compromise among the other constraints.

Automatic Garbage Collection

We classify automatic memory reclamation algorithms into two classes:

- reference counters: each allocated region knows how many references there are to it. When this number becomes zero, the region is freed.
- sweep algorithms: starting from a set of *roots*, the collection of all accessible values is traversed in a way similar to the traversal of a directed graph.

Sweep algorithms are more commonly used in programming languages. In effect, reference counting garbage collectors increase the processing costs (through counter updating) even when there is no need to reclaim anything.

Reference Counting

Each allocated region (object) is given a counter. This counter indicates the number of pointers to the object. It is incremented each time a reference to the object is shared. It is decremented whenever a pointer to the object disappears. When the counter becomes zero, the object is garbage collected.

The advantage of such a system comes from the immediate freeing of regions that are no longer used. Aside from the systematic slowdown of computations, reference counting garbage collectors suffer from another disadvantage: they do not know how to process circular objects. Suppose that Objective Caml had such a mechanism. The following example constructs a temporary value `l`, a list of characters of where the last element points to the cell containing `'c'`. This is clearly a circular value (figure 9.2).

```
# let rec l = 'c' :: 'a' :: 'm' :: l in List.hd l ;;
- : char = 'c'
```

At the end of the calculation of this expression each element of the list `l` has a counter

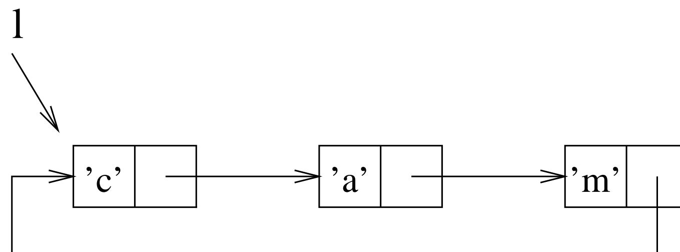

Figure 9.2: Memory representation of a circular list.

equal to one (even the first element, for the tail points to the head). This value is no longer accessible and yet cannot be reclaimed because its reference counter is not zero. In languages equipped with memory reclamation via reference counting—such as Python—and which allow the construction of circular values, it is necessary to add a memory sweep algorithm.

Sweep Algorithms

Sweep algorithms allow us to explore the graph of accessible values on the heap. This exploration uses a set of roots indicating the beginning of the traversal. These roots are exterior to the heap, stored most often in a stack. In the example in figure 9.1, we can suppose that the values of `u` and `v` are roots. The traversal starting from these roots constructs the graph of the values to save: the cells and pointers marked with heavy lines in figure 9.3.

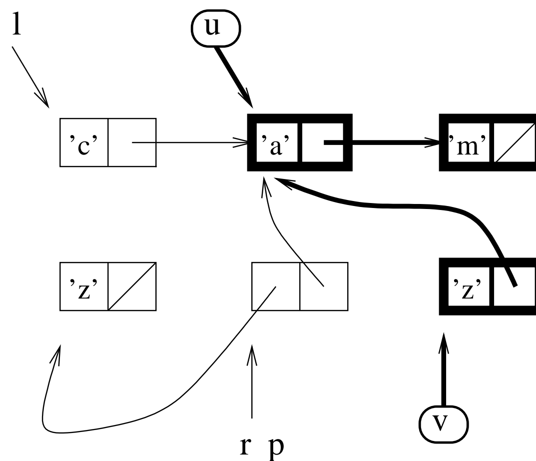

Figure 9.3: Memory reclamation after a garbage collection.

The traversal of this graph necessitates knowing how to distinguish immediate values from pointers in the heap. If a root points to an integer, we must not consider this value to be the address of another cell. In functional languages, this distinction is made by using a few bits of each cell of the heap. We call these bits *tag bits*. This is why Objective Caml integers only use 31 bits. This option is described in Chapter 12, page 325. We describe other solutions to the problem of distinguishing between pointers and immediate values in this chapter, page 260.

The two most commonly used algorithms are *Mark&Sweep*, which constructs the list of the free cells in the heap, and *Stop&Copy*, which copies cells that are still alive to a second memory region.

The heap should be seen as a vector of memory boxes. The representation of the state of the heap for the example of figure 9.1 is illustrated in figure 9.4.

We use the following characteristics to evaluate a sweep algorithm:

- efficiency: does the time-complexity depend on the size of the heap or only on the number of the living cells?
- reclamation factor: is all of the free memory usable?
- compactness: is all of the free memory usable in a single block?

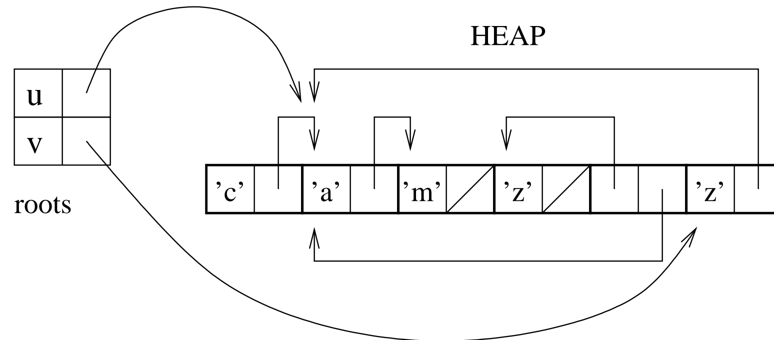

Figure 9.4: State of the heap.

- localization: are all of the different cells of a structured value close to one another?
- memory needs: does the algorithm need to use part of the memory when it runs?
- relocation: do values change location following a garbage collection?

Localization avoids changing memory pages when traversing a structured value. Compactness avoids fragmentation of the heap and allows allocations equal to the amount of available memory. The efficiency, reclamation factor, and supplementary memory needs are intimately linked to the time and space complexity of the algorithm.

Mark&*Sweep*

The idea of *Mark*&*Sweep* is to keep an up-to-date list of the free cells in the heap called the free list. If, at the time of an allocation request, the list is empty or no longer contains a free cell of a sufficient size, then a *Mark*&*Sweep* occurs.

It proceeds in two stages:

1. the marking of the memory regions in use, starting from a set of roots (called the *Mark* phase); then
2. reclamation of the unmarked memory regions by sequentially sweeping through the whole heap (called the *Sweep* phase).

One can illustrate the memory management of *Mark*&*Sweep* by using four "colorings" of the heap cells: white, gray[1], black, and hached. The mark phase uses the gray; the sweep phase, the hached; and the allocation phase, the white.

The meaning of the gray and black used by marking is as follows:

- gray: marked cells whose descendents are not yet marked;
- black: marked cells whose descendents are also marked.

1. In the online version of the book, the gray is slightly bluish.

It is necessary to keep the collection of grayed cells in order to be sure that everything has been explored. At the end of the marking each cell is either white or black, with black cells being those that were reached from the roots. Figure 9.5 shows an intermediate marking stage for the example of figure 9.4: the root `u` has been swept, and the sweeping of `v` is about to begin.

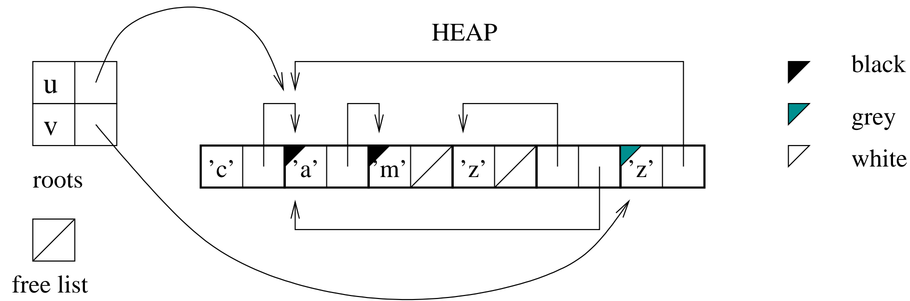

Figure 9.5: Marking phase.

It's during the sweep phase that the free list is constructed. The sweep phase modifies the colorings as follows:

- black becomes white, as the cell is alive;
- white becomes hached, and the cell is added to the free list.

Figure 9.6 shows the evolution of the colors and the construction of the free list.

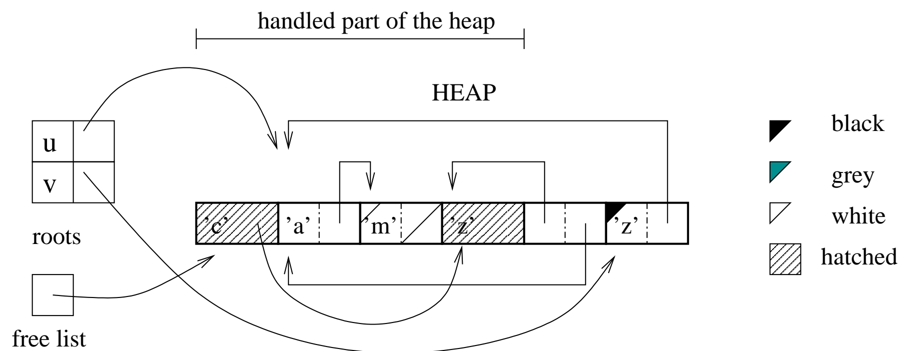

Figure 9.6: Sweep phase.

Characteristics of *Mark*&*Sweep* are that it:

- depends on the size of the entire heap (*Sweep* phase);
- reclaims all possible memory;
- does not compact memory;

- does not guarantee localization;
- does not relocate data.

The marking phase is generally implemented by a recursive function, and therefore uses space on the execution stack. One can give a completely iterative version of *Mark&Sweep* that does not require a stack of indefinite size, but it turns out to be less efficient than the partially recursive version.

Finally, *Mark&Sweep* needs to know the size of values. The size is either encoded in the values themselves, or deduced from the memory address by splitting the heap into regions that allocate objects of a bounded size. The *Mark&Sweep* algorithm, implemented since the very first versions of Lisp, is still widely used. A part of the Objective Caml garbage collector uses this algorithm.

Stop&Copy

The principal idea of this garbage collector is to use a secondary memory in order to copy and compact the memory regions to be saved. The heap is divided into two parts: the useful part (called *from-space*), and the part being re-written (called *to-space*).

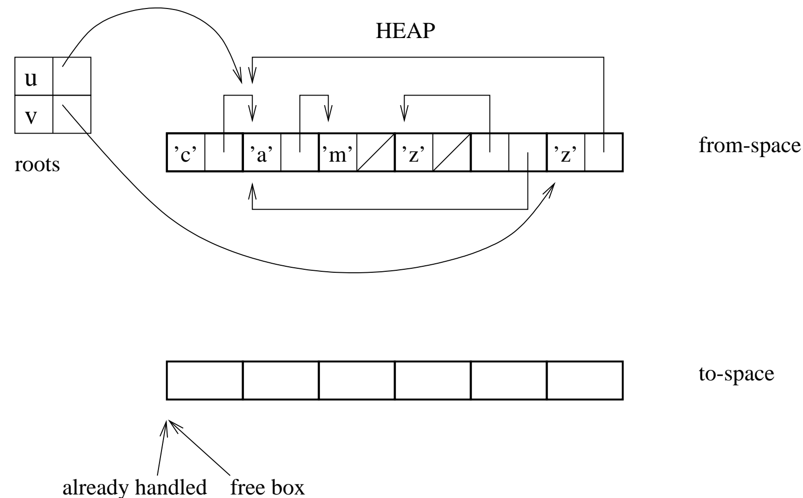

Figure 9.7: Beginning of *Stop&Copy*.

The algorithm is the following. Beginning from a set of roots, each useful part of the *from-space* is copied to the *to-space*; the new address of a relocated value is saved (most often in its old location) in order to update all of the other values that point to this value.

The contents of the rewritten cells gives new roots. As long as there are unprocessed roots the algorithm continues.

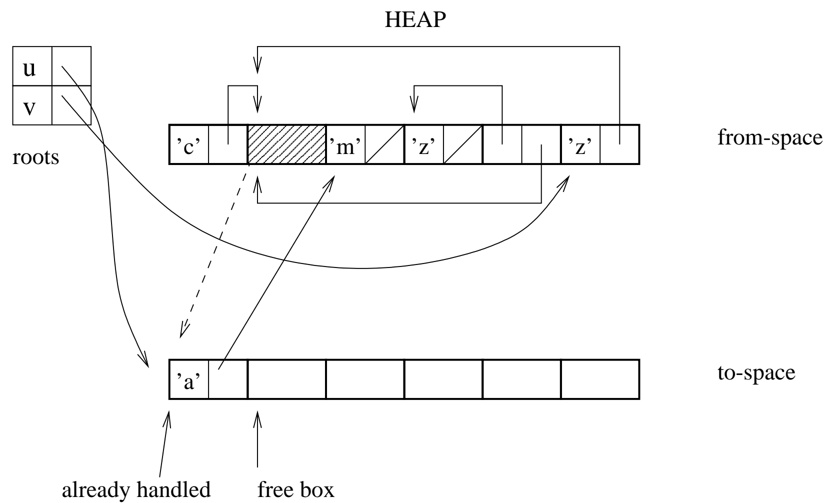

Figure 9.8: Rewriting from *from-space* into *to-space*.

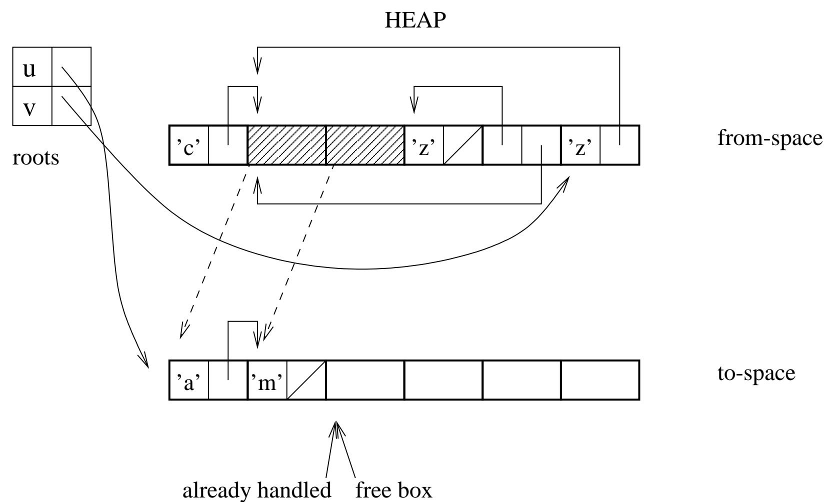

Figure 9.9: New roots.

In the case of sharing, in other words, when attempting to relocate a value that has already been relocated, it suffices to use the new address.

At the end of garbage collection, all of the roots are updated to point to their new addresses. Finally, the roles of the two parts are reversed for the next garbage collection.

The principal characteristics of this garbage collector are the following:

- it depends solely on the size of the objects to be kept;

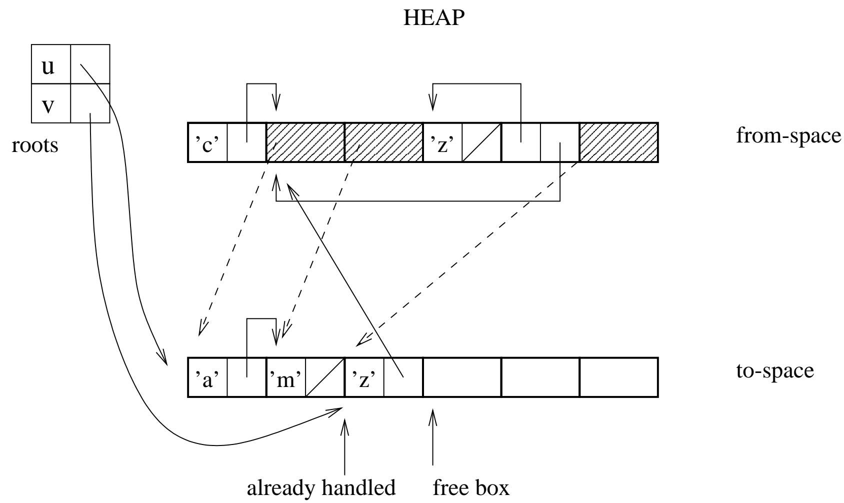

Figure 9.10: Sharing.

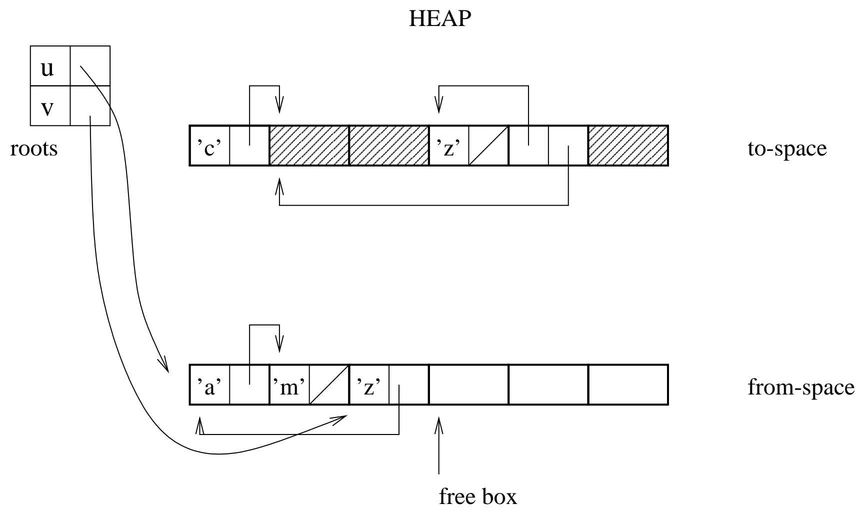

Figure 9.11: Reversing the two parts.

- only half of the memory is available;
- it compacts memory;
- it may localize values (using breadth-first traversal);
- it does not use extra memory (only *from-space*+*to-space*);
- the algorithm is not recursive;
- it relocates values into the new part of memory;

Other Garbage Collectors

Many other techniques, often derived from the two preceding, have been used: either in particular applications, *e.g.,* the manipulation of large matrices in symbolic calculations, or in a general way linked to compilation techniques. Generational garbage collectors allow optimizations based on the age of the values. Conservative garbage collectors are used where there is not an explicit differentiation between immediate values and pointers (for example, when one translates into C). Finally, incremental garbage collectors allow us to avoid a noticeable slow-down at the time of garbage collection activation.

Generational Garbage Collection

Functional programs are, in general, programs that allocate frequently. We notice that a very large number of values have a very short lifetime[2]. On the other hand, when a value has survived several garbage collections, it is quite likely to survive for a while longer. In order to avoid complete traversal of the heap—as in *Mark&Sweep*—during each memory reclamation, we would like to be able to traverse only the values that have survived one or more garbage collections. Most frequently, it is among the young values that we will recover the most space. In order to take advantage of this property, we give objects dates, either a time-stamp or the number of garbage collections survived. To optimize garbage collection, we use different algorithms according to the age of the values:

- The garbage collections for young objects should be fast and traverse only the younger generations.
- The garbage collections for old objects should be rare and do well at collecting free space from the entire memory.

As a value ages it should take part less and less in the most frequent garbage collections. The difficulty, therefore, is taking count of only the region of memory occupied by young objects. In a purely functional language, that is, a language without assignment, younger objects reference older objects, and on the other hand, older objects do not possess pointers to younger objects because they were created before the young objects existed. Therefore, these garbage collection techniques lend themselves well to functional languages, with the exception of those with delayed evaluation which can in fact evaluate the constituents of a structure after evaluating the structure itself. On the other hand, for functional languages with assignment it is always possible to modify part of an older object to refer to a younger object. The problem then is to save young memory regions referenced only by an older value. For this, it is necessary to keep an up-to-date table of references from old objects to young objects in order to have a correct garbage collection. We study the case of Objective Caml in the following section.

2. Most values do not survive a single garbage collection.

Conservative Garbage Collectors

To this point, all of the garbage collection techniques presume knowing how to tell a pointer from an immediate value. Note that in functional languages with parametric polymorphism values are uniformly represented, and in general occupy one word of memory[3]. This is what allows having generic code for polymorphic functions.

However, this restriction on the range for integers may not be acceptable. In this case, conservative garbage collectors make it possible to avoid marking immediate values such as integers. In this case, every value uses an entire memory word without any tag bits. In order to avoid traversing a memory region starting from a root actually containing an integer, we use an algorithm for discriminating between immediate values and pointers that relies on the following observations:

- the addresses of the beginning and end of the heap are known so any value outside of these bounds is an immediate value;
- allocated objects are aligned on a word address. Every value that does not correspond to such an alignment must also be an immediate value.

Thus each heap value that is valid from the point of view of being an address into the heap is considered to be a pointer and the garbage collector tries to keep this region, including those cases where the value is in fact an immediate value. These cases may become very rare by using specific memory pages according to the size of the objects. It is not possible to guarantee that the entire unused heap is collected. This is the principal defect of this technique. However, we remain certain that only unused regions are reclaimed.

In general, conservative garbage collectors are conservative, *i.e.,* they do not relocate objects. Indeed, as the garbage collector considers some immediate values as pointers, it would be harmful to change their value. Nevertheless, some refinements can be introduced for building the sets of roots, which allow to relocate corresponding to clearly known roots.

Garbage collection techniques for ambiguous roots are often used when compiling a functional language into C, seen here as a portable assembler. They allow the use of immediate C values coded in a memory word.

Incremental Garbage Collection

One of the criticisms frequently made of garbage collection is that it stops the execution of a running program for a time that is perceptible to the user and is unbounded. The first is embarrassing in certain applications, for instance, rapid-action games where the halting of the game for a few seconds is too often prejudicial to the player, as the execution restarts without warning. The latter is a source of loss of control for applications which must process a certain number of events in a limited time. This is

3. The only exception in Objective Caml relates to arrays of floating point values (see chapter 12, page 331).

typically the case for embedded programs which control a physical device such as a vehicle or a machine tool. These applications, which are real-time in the sense that they must respond in a bounded time, most often avoid using garbage collectors.

Incremental garbage collectors must be able to be interrupted during any one of their processing phases and be able to restart while assuring the safety of memory reclamation. They give a sufficiently satisfactory method for dealing with the former case, and can be used in the latter case by enforcing a programming discipline that clearly isolates the software components that use garbage collection from those that do not.

Let us reconsider the *Mark&Sweep* example and see what adaptations are necessary in order to make it incremental. There are essentially two:

1. how to be sure of having marked everything during the marking phase?
2. how to allocate during either the marking phase or the reclamation phase?

If *Mark&Sweep* is interrupted in the *Mark* phase, it is necessary to assure that cells allocated between the interruption of marking and its restart are not unduly reclaimed by the *Sweep* that follows. For this, we mark cells allocated during the interruption in black or gray in anticipation.

If the *Mark&Sweep* is interrupted during the *Sweep* phase, it can continue as usual in re-coloring the allocated cells white. Indeed, as the *Sweep* phase sequentially traverses the heap, the cells allocated during the interruption are localized before the point where the sweep restarts, and they will not be re-examined before the next garbage collection cycle.

Figure 9.12 shows an allocation during the reclamation phase. The root `w` is created by:

```
# let w = 'f' :: v;;
val w : char list = ['f'; 'z'; 'a'; 'm']
```

Memory Management by Objective Caml

Objective Caml's garbage collector combines the various techniques described above. It works on two generations, the old and the new. It mainly uses a *Stop&Copy* on the new generation (a minor garbage collection) and an incremental *Mark&Sweep* on the old generation (major garbage collection).

A young object that survives a minor garbage collection is relocated to the old generation. The *Stop&Copy* uses the old generation as the `to-space`. When it is finished, the entire `from-space` is completely freed.

When we presented generational garbage collectors, we noted the difficulty presented by impure functional languages: an old-generation value may reference an object of the new generation. Here is a small example.

```
# let older = ref [1] ;;
```

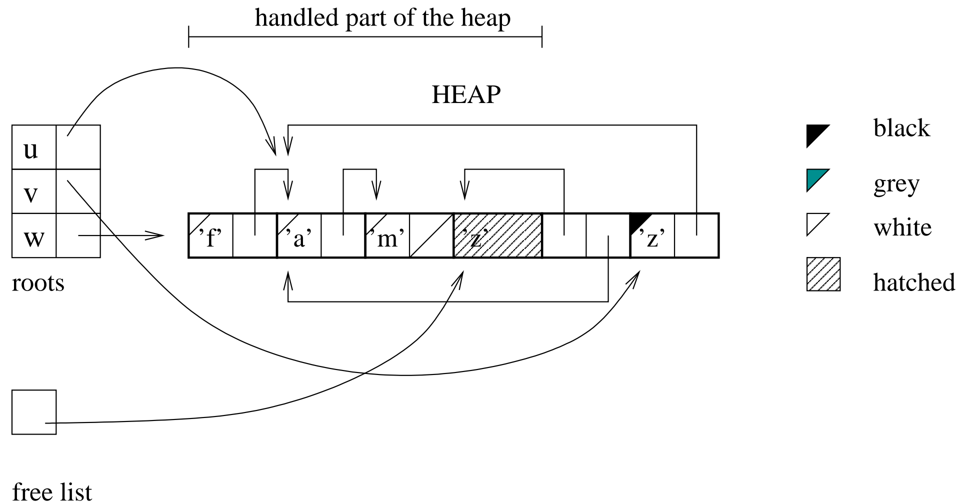

Figure 9.12: Allocation during reclamation.

```
val older : int list ref = {contents=[1]}
(* ... *)
# let newer = [2;5;8] in
    older := newer ;;
- : unit = ()
```

The comment `(* ... *)` replaces a long sequence of code in which `older` passes into the older generation. The minor garbage collection must take account of certain old generation values. Therefore we must keep an up-to-date table of the references from the old generation to the new that becomes part of the set of roots for the minor garbage collection. This table of roots grows very little and becomes empty just after a minor garbage collection.

It is to be noted that the *Mark&Sweep* of the old generation is incremental, which means that a part of the major garbage collection happens during each minor garbage collection. The major garbage collection is a *Mark&Sweep* that follows the algorithm presented on page 259. The relevance of this incremental approach is the reduction of waiting time for a major garbage collection by advancing the marking phase with each minor garbage collection. When a major garbage collection is activated, the marking of the unprocessed regions is finished, and the reclamation phase is begun. Finally, as *Mark&Sweep* may fragment the old generation significantly, a compaction algorithm may be activated after a major garbage collection.

Putting this altogether, we arrive at the following stages:

1. minor garbage collection: perform a *Stop&Copy* on the young generation; age the surviving objects by having them change zone; and then do part of the *Mark&Sweep* of the old generation.
   It fails if the zone change fails, in which case we go to step 2.

2. end of the major garbage collection cycle.
   When this fails go on to step 3.
3. another major garbage collection, to see if the objects counted as used during the incremental phases have become free.
   When this fails, go on to step 4.
4. Compaction of the old generation in order to obtain maximal contiguous free space. If this last step does not succeed, there are no other possibilities, and the program itself fails.

The GC module allows activation of the various phases of the garbage collector.

A final detail of the memory management of Objective Caml is that the heap space is not allocated once and for all at the beginning of the program, but evolves with time (increasing or decreasing by a given size).

Module Gc

The Gc module lets one obtain statistics about the heap and gives control over its evolution as well as allowing the activation of various garbage collector phases. Two concrete record types are defined: *stat* and *control*. The fields of type *control* are modifiable; whereas those of *stat* are not. The latter simply reflect the state of the heap at a given moment.

The fields of a *stat* mainly contain counters indicating:

- the number of garbage collections: minor_collections, major_collections and compactions;
- the number of words allocated and transfered since the beginning of the program: minor_words, promoted_words, and major_words.

The fields of the record *control* are:

- minor_heap_size, which defines the size of the zone allotted to the younger generation;
- major_heap_increment, which defines the increment applied to the growth of the region for the older generation;
- space_overhead, which defines the percentage of the memory used beyond which a major garbage collection is begun (the default value is 42);
- max_overhead, which defines the connection between free memory and occupied memory after which compactification is activated. A value of 0 causes a systematic compactification after every major garbage collection. The maximal value of 1000000 inhibits compactification.
- verbose is an integer parameter governing the tracing of the activities of the garbage collector.

Functions manipulating the types *stat* and *control* are given in figure 9.13.

stat	*unit* → *stat*
print_stat	*out_channel* → *unit*
get	*unit* → *control*
set	*control* → *unit*

Figure 9.13: Control and statistical functions for the heap.

The following functions, of type *unit -> unit*, force the execution of one or more stages of the Objective Caml garbage collector: `minor` (stage 1), `major` (stages 1 and 2), `full_major` (stages 1, 2 and 3) and `compact` (stages 1, 2, 3 and 4).

Examples

Here is what the `Gc.stat` call shows:

```
# Gc.stat();;
- : Gc.stat =
{Gc.minor_words=555677; Gc.promoted_words=61254; Gc.major_words=205249;
 Gc.minor_collections=17; Gc.major_collections=3; Gc.heap_words=190464;
 Gc.heap_chunks=3; Gc.live_words=157754; Gc.live_blocks=35600;
 Gc.free_words=32704; Gc.free_blocks=83; Gc.largest_free=17994;
 Gc.fragments=6; Gc.compactions=0}
```

We see the number of executions of each phase: minor garbage collection, major garbage collection, compaction, as well as the number of words handled by the different memory spaces. Calling `compact` forces the four stages of the garbage collector, causing the heap statistics to be modified (see the call of `Gc.stat`).

```
# Gc.compact();;
- : unit = ()
# Gc.stat();;
- : Gc.stat =
{Gc.minor_words=562155; Gc.promoted_words=62288; Gc.major_words=206283;
 Gc.minor_collections=18; Gc.major_collections=5; Gc.heap_words=190464;
 Gc.heap_chunks=3; Gc.live_words=130637; Gc.live_blocks=30770;
 Gc.free_words=59827; Gc.free_blocks=1; Gc.largest_free=59827;
 Gc.fragments=0; Gc.compactions=1}
```

The fields `GC.minor_collections` and `compactions` are incremented by 1, whereas the field `Gc.major_collections` is incremented by 2. All of the fields of type *GC.control* are modifiable. For them to be taken into account, we must use the function `Gc.set`, which takes a value of type *control* and modifies the behavior of the garbage collector.

For example, the field `verbose` may take a value from 0 to 127, controlling 7 different indicators.

```
# c.Gc.verbose <- 31;;
```

```
Characters 1-2:
This expression has type int * int but is here used with type Gc.control
# Gc.set c;;
Characters 7-8:
This expression has type int * int but is here used with type Gc.control
# Gc.compact();;
- : unit = ()
```

which prints:

```
<>Starting new major GC cycle
allocated_words = 329
extra_heap_memory = 0u
amount of work to do = 3285u
Marking 1274 words
!Starting new major GC cycle
Compacting heap...
done.
```

The different phases of the garbage collector are indicated as well as the number of objects processed.

Module **Weak**

A weak pointer is a pointer to a region which the garbage collector may reclaim at any moment. It may be surprising to speak of a value that might disappear at any moment. In fact, we must see these weak pointers as a reservoir of values that may still be available. This turns out to be particularly useful when memory resources are small compared to the elements to be saved. The classic case is the management of a memory cache: a value may be lost, but it remains directly accessible as long as it exists.

In Objective Caml one cannot directly manipulate weak pointers, only arrays of weak pointers. The `Weak` module defines the abstract type *'a Weak.t*, corresponding to the type *'a option array*, a vector of weak pointers of type `'a`. The concrete type *'a option* is defined as follows:

```
type 'a option = None | Some of 'a;;
```

The main functions of this module are defined in figure 9.14.

The `create` function allocates an array of weak pointers, each initialized to `None`. The `set` function puts a value of type *'a option* at a specified index. The `get` function returns the value contained at index n in a table of weak pointers. The returned value is then referenced, and no longer reclaimable as long as this reference exists. To

function	type
create	*int -> 'a t*
set	*'a t -> int -> 'a option -> unit*
get	*'a t -> int -> 'a option*
check	*'a t -> int -> bool*

Figure 9.14: Main functions of the `Weak` module.

verify the effective existence of a value, one uses either the `check` function or pattern matching on the *'a option* type's patterns. The former solution does not depend on the representation choice for weak pointers.

Standard functions for sequential structures also exist: `length`, for the length, and `fill` and `blit` for copies of parts of the array.

Example: an Image Cache

In an image-processing application, it is not rare to work on several images. When the user moves from one image to another, the first is saved to a file, and the other is loaded from another file. In general, only the names of the latest images processed are saved. In order to avoid overly frequent disk access while at the same time not using too much memory space, we use a memory cache which contains the last images loaded. The contents of the cache may be freed if necessary. We implement this with a table of weak pointers, leaving the decision of when to free the images up to the garbage collector. To load an image we first search the cache. If the image is there, it becomes the current image. If not, its file is read.

We define a table of images in the following manner:

```
# type table_of_images = {
    size : int;
    mutable ind : int;
    mutable name : string;
    mutable current : Graphics.color array array;
    cache : ( string * Graphics.color array array) Weak.t } ;;
```

The field `size` gives the size of the table; the field `ind` gives the index of the current image; the field `name`, the name of the current image; the field `current`, the current image, and the field `cache` contains the array of weak pointers to the images. It contains the last images loaded and their names.

The function `init_table` initializes the table with its first image.

```
# let open_image filename =
    let ic = open_in filename
    in let i = ((input_value ic) : Graphics.color array array)
    in ( close_in ic ; i ) ;;
val open_image : string -> Graphics.color array array = <fun>
```

```
# let init_table n filename =
    let i = open_image filename
    in let c =  Weak.create n
    in Weak.set c 0 (Some (filename,i)) ;
       { size=n; ind=0; name = filename; current = i; cache = c } ;;
val init_table : int -> string -> table_of_images = <fun>
```

The loading of a new image saves the current image in the table and loads the new one. To do this, we must first try to find the image in the cache.

```
# exception Found of int * Graphics.color array array ;;
# let search_table filename table =
    try
      for i=0 to table.size-1 do
        if i<>table.ind then match Weak.get table.cache i with
              Some (n,img) when n=filename  → raise (Found (i,img))
            | _ → ()
        done ;
      None
    with Found (i,img) → Some (i,img) ;;
```

```
# let load_table filename table =
    if table.name = filename then ()  (* the image is the current image *)
    else
      match search_table filename table with
        Some (i,img) →
        (* the image found becomes the current image *)
          table.current <- img ;
          table.name <- filename ;
          table.ind <- i
      | None →
        (* the image isn't in the cache, need to load it *)
        (* find an empty spot in the cache *)
          let i = ref 0 in
            while (!i<table.size && Weak.check table.cache !i) do incr i done ;
            (* if none are free, take a full slot *)
              ( if !i=table.size then i:=(table.ind+1) mod table.size ) ;
              (* load the image here and make it the current one *)
              table.current <- open_image filename ;
              table.ind <- !i ;
              table.name <- filename ;
              Weak.set table.cache table.ind (Some (filename,table.current)) ;;
val load_table : string -> table_of_images -> unit = <fun>
```

The `load_table` function tests to see if the image requested is current. If not, it checks the cache to see if the image exists; if that fails, the function loads the image from disk. In either of the latter two cases, it makes the image become the current one.

To test this program, we use the following cache-printing function:

```
# let print_table table =
    for i = 0 to table.size-1 do
      match Weak.get table.cache ((i+table.ind) mod table.size) with
        None → print_string "[] "
      | Some (n,_) → print_string n ; print_string " "
    done ;;
val print_table : table_of_images -> unit = <fun>
```

Then we test the following program:

```
# let t = init_table 10 "IMAGES/animfond.caa" ;;
val t : table_of_images =
  {size=10; ind=0; name="IMAGES/animfond.caa";
   current=
    [|[|7372452; 7372452; 7372452; 7372452; 7372452; 7372452; 7372452;
        7372452; 7372452; 7372452; 7372452; 7372452; 7505571; 7505571; ...|];
      ...|];
   cache=...}
# load_table "IMAGES/anim.caa" t ;;
- : unit = ()
# print_table t ;;
IMAGES/anim.caa [] [] [] [] [] [] [] [] [] - : unit = ()
```

This cache technique can be adapted to various applications.

Exercises

Following the evolution of the heap

In order to follow the evolution of the heap, we suggest writing a function that keeps information on the heap in the form of a record with the following format:

```
# type tr_gc = {state : Gc.stat;
                time : float; number : int};;
```

The time corresponds to the number of milliseconds since the program began and the number serves to distinguish between calls. We use the function `Unix.time` (see chapter 18, page 572) which gives the running time in milliseconds.

1. Write a function `trace_gc` that returns such a record.
2. Modify this function so that it can save a value of type *tr_gc* in a file in the form of a persistant value. This new function needs an output channel in order to write. We use the `Marshal` module, described on page 228, to save the record.
3. Write a stand-alone program, taking as input the name of a file containing records of type of *tr_gc*, and displaying the number of major and minor garbage collections.

4. Test this program by creating a trace file at the interactive loop level.

Memory Allocation and Programming Styles

This exercise compares the effect of programming styles on the growth of the heap. To do this, we reconsider the exercise on prime numbers from chapter 8 page 244. We are trying to compare two versions, one tail-recursive and the other not, of the sieve of Eratosthenes.

1. Write a tail-recursive function `erart` (this name needs fixing) that calculates the prime numbers in a given interval. Then write a function that takes an integer and returns the list of smaller prime numbers.
2. By using the preceding functions, write a program (change the name) that takes the name of a file and a list of numbers on the command line and calculates, for each number given, the list of prime numbers smaller than it. This function creates a garbage collection trace in the indicated file. Trace commands from previous exercice are gathered in file `trgc.ml`
3. Compile these files and create a stand-alone executable; test it with the following call, and display the result.

```
                                                        %
erart trace_rt 3000 4000 5000 6000
```

4. Do the same work for the non tail recursive function.
5. Compare trace results.

Summary

This chapter has presented the principal families of algorithms for automatic memory reclamation with the goal of detailing those used in Objective Caml. The Objective Caml garbage collector is an incremental garbage collector with two generations. It uses *Mark&Sweep* for the old generation, and *Stop&Copy* for the young generation. Two modules directly linked to the garbage collector allow control of the evolution of the heap. The `Gc` module allows analysis of the behavior of the garbage collector and modification of certain parameters with the goal of optimizing specific applications. With the `Weak` module one can save in arrays values that are potentially reclaimable, but which are still accessible. This module is useful for implementing a memory cache.

To Learn More

Memory reclamation techniques have been studied for forty years—in fact, since the first implementations of the Lisp programming language. For this reason, the literature in this area is enormous.

A comprehensive reference is Jones' book [Jon98]. Paul Wilson's tutorial [Wil92] is an excellent introduction to the field, with many references. The following web pages also provide a good view of the state of the art in memory management.

Link: ftp://ftp.netcom.com/pub/hb/hbaker/home.html

is an introduction to sequential garbage collectors.

Link: http://www.cs.ukc.ac.uk/people/staff/rej/gc.html

contains the presentation of [Jon98] and includes a large searchable bibliography.

Link: http://www.cs.colorado.edu/~zorn/DSA.html

lists different tools for debugging garbage collection.

Link: http://reality.sgi.com/boehm_mti/

offers C source code for a conservative garbage collector for the C language. This garbage collector replaces the classical allocator `malloc` by a specialized version `GC_malloc`. Explicit recovery by `free` is replaced by a new version that no longer does anything.

Link: http://www.harlequin.com/mm/reference/links.html

maintains a list of links on this subject.

In chapter 12 on the interface between C and Objective Caml we come back to memory management.