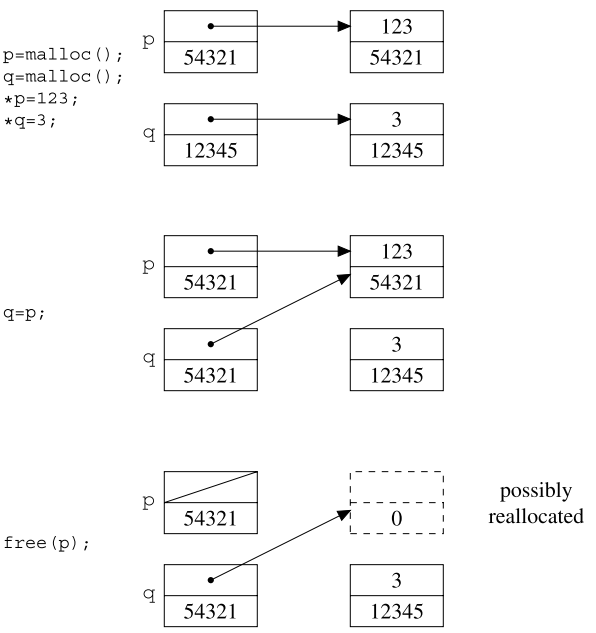
Fig. 8.11 Locks and keys



importantly, the high cost of pointer assignment and of determining whether the key opens a lock, something that happens every time a pointer is dereferenced.

8.11 Garbage Collection

In languages without explicit memory deallocation, it is necessary to equip the abstract machine with a mechanism which could automatically reclaim the memory allocated on the heap that is no longer used. This is done by a *garbage collector*, introduced for the first time in LISP around 1960, and since then included in many languages initially mostly functional, and later imperative. Java has a powerful and efficient garbage collector.

From the logical point of view, the operation of a garbage collector consists of two phases:

1. Distinguish those objects that are still alive from those no longer in use (*garbage detection*);
2. Collect those objects known no longer to be in use, so that the program can reuse them.

In practice these two phases are not always temporally separate, and the technique for reclaiming objects essentially depends on the one in which the objects no longer in use are discovered. We will see, then, that the "no longer in use" concept in a garbage collector is often a conservative approximation. For reasons of effi-

ciency,[18] not all the objects that can be used again are, in reality, determined by the garbage collector to be so.

The educational aim of this text does not permit us to describe a garbage collector for a real language, or to provide an exhaustive overview of the different techniques available (there are bibliographic references to exhaustive treatments). We will limit ourselves to presenting in some detail the main points in the light of the most common techniques. Real garbage collectors are variations on these techniques. In particular the collectors that are of greatest interest today are based upon some form of incremental reclamation of memory and are beyond the scope of this book.

We can classify classical garbage collectors according to how they determine whether an object is no longer in use. We will have, therefore, collectors based on *reference counting*, as well as *marking* and *copying*. In the next few sections, we will present these collectors. We will discuss these techniques in terms of pointers and objects but the argument can equally be applied to languages without pointers which use the reference variable model.

Finally, we will see that all the techniques that we consider need to be able to recognize those fields inside an object that correspond to pointers. If objects are created as instances of statically-defined types (so it is statically known where pointers are inside instances), the compiler can generate a descriptor containing offsets to the pointers in each type. Each object in the heap is associated with the type of which it is an instance (for example, via a pointer to its type descriptor). When an object is to be deallocated, the garbage collector accesses the type descriptor to access the pointers inside the object. Similar techniques are used to recognise which words in an activation record are pointers. If types are only known dynamically, descriptors have to be completely dynamic and allocated together with the object.

8.11.1 Reference Counting

A simple way to tell whether an object is not in use is to determine whether there are any pointers to it. This is the reference counting technique. It defines what is probably the easiest way of implementing a garbage collector.

When an object is created on the heap, an integer is allocated at the same time. This integer is the *reference counter* or *reference count* for the object. The reference count is inaccessible to the programmer. For each object, the abstract machine must ensure that the counter contains the number of pointers to this object that are still active.

In order to do this, when an object is created (that is, a pointer for it is assigned for the first time), its counter is initialised to 1. When we have a pointer assignment:

```
p = q;
```

[18] As well as decidability.

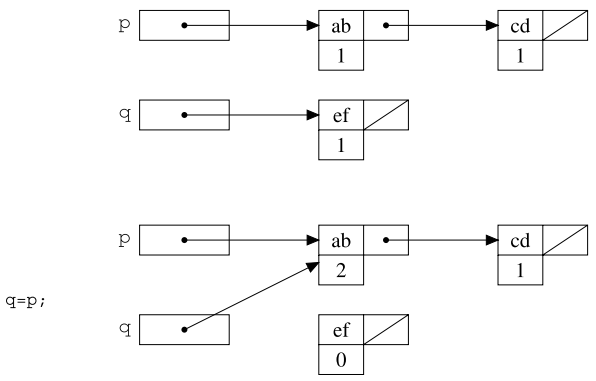


Fig. 8.12 Reference counting

the counter belonging the object pointed to by `q` is incremented by one, while the counter of object pointed to by `p` is decremented by one. When a local environment is exited, the counters of all the objects pointed to by pointers local to the environment are decremented. Figure 8.12 shows operation of this technique in diagrammatic form.

When a counter returns to the value 0, its associated object can be deallocated and returned to the free list. This object, moreover, might contain internal pointers, so before returning the memory occupied by an object to the free list, the abstract machine follows the pointers inside the object and decrements by one the counters of the objects they point to, recursively collecting all the objects whose counter has reached 0.

From the viewpoint of the abstract division into two phases of a garbage collector, the update and checking of counters implement the garbage-detection phase. The collection phase occurs when the counter reaches zero.

A clear advantage of this technique is that it is *incremental* because checking and collection are interleaved with the normal operation of the program. With a few adjustments, real-time systems (that is systems where there are absolute deadlines to response time) can employ this technique.

The biggest defect, at least in principle, of this technique is in its inability to deallocate circular structures. Figure 8.13 shows a case in which a circular structure has no more access paths. However it cannot be collected because, clearly, its counters are not zero. It can be seen that the problem does not reside so much in the algorithm, as in the definition of what a useless object is. It is clear that all objects in a circular structure are not usable any more, but this is not at all captured by the definition of not being pointed at.

Reference counting, despite its simplicity, is also fairly inefficient. Its cost is proportional to the combined work performed by the program (and not to the size of the heap or to the percentage of it in use or not in use at any time). One particular case is that of updating the counters of parameters of pointer type which are passed to functions that execute only for a short time. These counters are allocated and, after a particularly short time, have their value returned to zero.

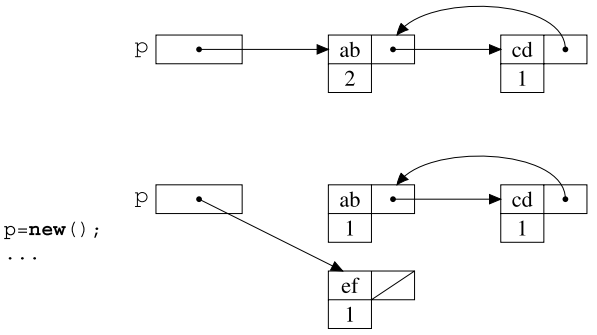


Fig. 8.13 Circular structures with reference counts

8.11.2 Mark and Sweep

The *mark and sweep* method takes its name from the way in which the two abstract phases we mentioned at the start are implemented:

- *Mark*. To recognize that something is unused, all the objects in the heap are traversed, marking each of them has "not in use". Then, starting from the pointers which are active and present on the stack (the *root set*), all the data structures present in the heap are traversed recursively (the search is usually performed depth-first) and every object that is encountered is marked as "in use".
- *Sweep*. The heap is swept—all blocks marked "in use" are left alone, while those marked "not in use" are returned to the free list.

It can be seen that to implement both phases, it is necessary to be able to recognize allocated blocks in the heap. Descriptors might be necessary to give the size (and organisation) of every allocated block.

Unlike the reference-counting garbage collector, a mark and sweep collector is not incremental. It it will be invoked only when the free memory available in the heap is close to being exhausted. The user of the program can therefore experience a significant degradation in response time while waiting for the garbage collector to finish.

The mark and sweep technique suffers from three main defects. In the first place, and this is also true for reference counting, it is asymptotically the cause of external fragmentation (see Sect. 5.4.2): live and no longer live objects are arbitrarily mixed in the heap which can make allocating a large object difficult, even if many small blocks are available. The second problem is efficiency. It requires time proportional to the total length of the heap, independent of the percentages of used and free space. The third problem relates to locality of reference. Objects that are "in use" remain in their place, but it is possible that objects contiguous with them will be collected and new objects allocated in their place. During execution, we find that objects with very different ages appear next to each other. This, in general, drastically reduces locality of reference, with the usual degradation in performance observed in systems with memory hierarchies.

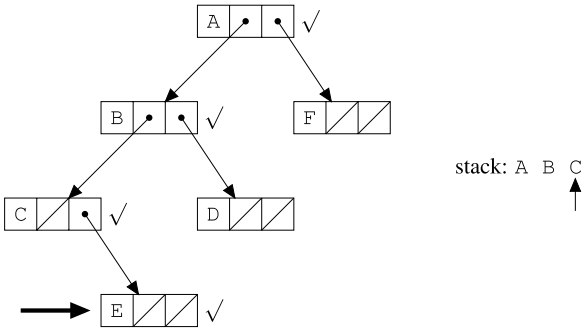


Fig. 8.14 A stack is necessary for depth-first traversal

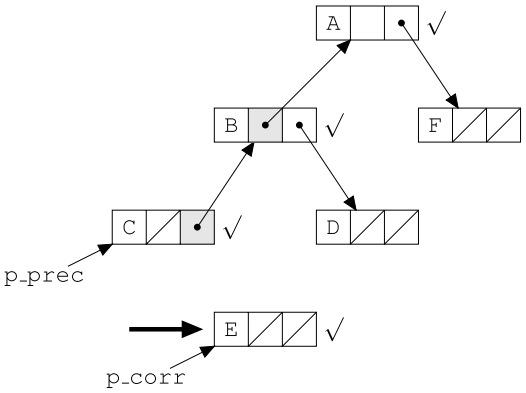
8.11.3 Interlude: Pointer Reversal

Without some precautions, every marking technique runs the risk of being completely unusable in a garbage collector. The collector, indeed, goes into action when the memory is near to being exhausted, while the marking phase consists of the recursive traversal of a graph, which requires the use of a stack to store the return pointers.[19] It is necessary, when marking a graph under these circumstances, to use carefully the unused space present in pointers, using a technique called *pointer reversal*.

As shown in Fig. 8.14, when visiting a chained structure, it is necessary to mark a node and recursively visit the substructures whose pointers are part of that node. In this recursive scheme, it is necessary to store the addresses of blocks when they are visited in a stack, so that they can be revisited when the end of the structure is reached. In the Figure, using depth-first search, node E has been reached (marked blocks are indicated with a tick). The stack contains nodes A, B and C. After visiting E, it takes C from the stack and follows any pointers leading from this node; since there are none, it takes B from the stack and follows the remaining pointer, pushing B onto the stack, and visiting D. Using pointer reversal, this stack is stored in the pointers that form the structure, as shown in Fig. 8.15. When the end of a substructure is reached, (and a pop of the stack is required), the pointer is returned to its original value, so that at the end of the visit, the structure is exactly the same as it was at the start (apart from marking, clearly). In the figure, we are visiting node E. It can be seen how the stack in Fig. 8.14 is stored in space internal to the structure itself (pointers marked with a grey background). Only two pointers (`p_prec` and `p_curr`) are required to perform the visit. After visiting E, using `p_prec`, the structure is retraversed in a single step. Using the reversed pointer, we return to B, resetting the correct value of the pointer in C using `p_curr`.

[19] In many abstract machines, stack and heap are implemented in a single area of memory, with stack and heap growing in opposite directions starting from the two ends. In such a case, when there is no space in the heap, there is no space for the stack.

Fig. 8.15 Pointer reversal



8.11.4 Mark and Compact

To avoid the fragmentation caused by the mark and sweep technique, we can modify the sweep phase and convert it into a compaction phase. Live objects are moved so that they are contiguous and thereby leave a contiguous block of free memory. Compaction can be performed by moving linearly across the heap, moving every live block encountered and making it contiguous with the previous block. At the end, all free blocks are contiguous, as are all unused blocks.

This is a technique which, like mark and sweep, requires more than one pass over the heap (and the time required is therefore proportional to the heap size). Compaction, on its own, requires two or three passes. The first is to compute the new position to be taken by the live blocks; a second updates the internal pointers and a third actually moves objects. It is, therefore, a technique that is substantially more expensive than mark and sweep if there are many objects to be moved.

On the other hand, compaction has optimal effect on fragmentation and on locality. It supports treating the free list as a single block. New objects are allocated from the free list by a simple pointer subtraction.

8.11.5 Copy

In garbage collectors based on copy there is no phase during which the "garbage" is marked. There is just the copying and compaction of live blocks. The lack of an explicit mark phase and the completely different way space is handled make its costs substantially different from those of algorithms based on marking.

In the simplest copy-base garbage collector (called a *stop and copy* collector), the heap is divided into two equally-sized parts (two *semi-spaces*). During normal execution, only one of the two semi-spaces is in use. Memory is allocated at one end of the semi-space, while free memory consists of a unique contiguous block which reduces its size every time there is an allocation (see Fig. 8.16). Allocation is extremely efficient and there is no fragmentation.

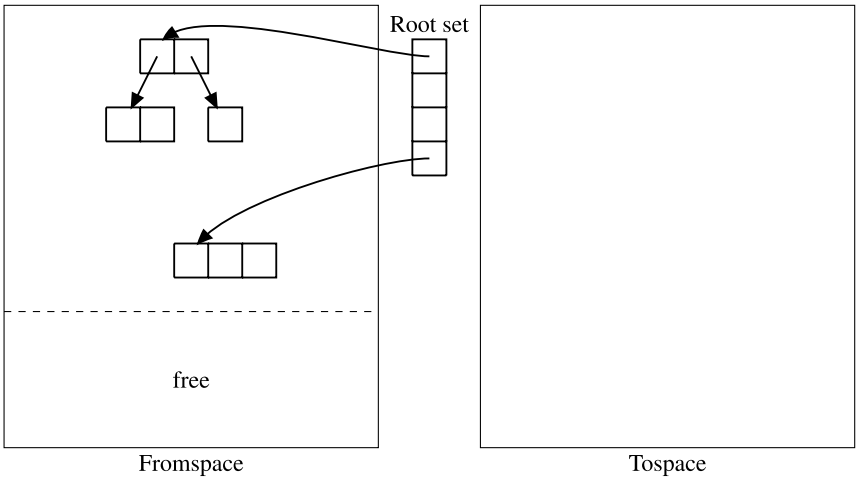


Fig. 8.16 Stop and copy before a call to the garbage collector

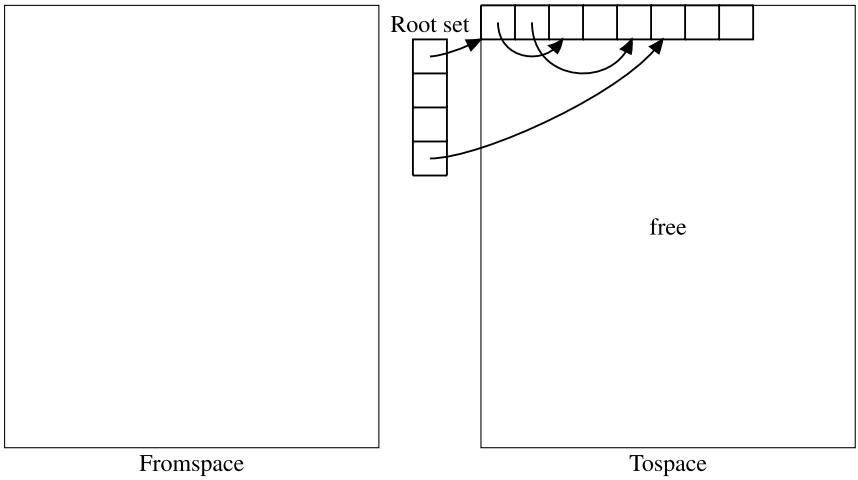


Fig. 8.17 Stop and copy after the execution of the garbage collector

When the memory in the semi-space is exhausted, the garbage collector is invoked. Starting with pointers in the stack (the root set), it begins visiting the chain of structures held in the current semi-space (the *fromspace*), copying the structures one by one into the other semi-space (the *tospace*), compacting them at one end of the latter (see Fig. 8.17). At the end of this process, the role of the two semi-spaces is swapped and execution returns to the user program.

The visit and copy of the live part can be executed in an efficient manner using the simple technique known as Cheney's algorithm (Fig. 8.18). Initially, all the objects which are immediately reachable from the root set are copied. This first set of

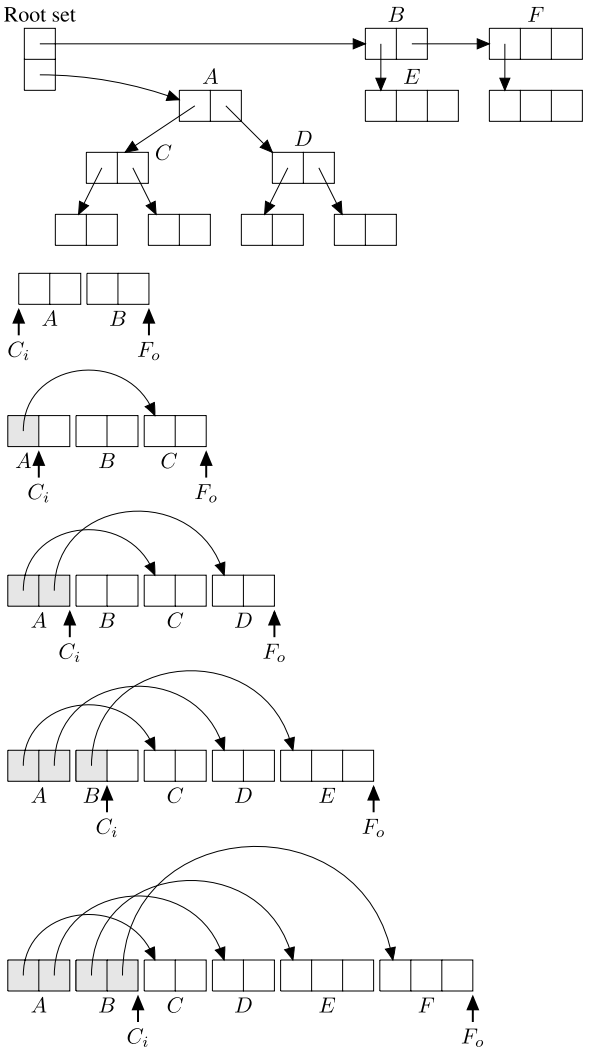


Fig. 8.18 Cheney's algorithm

objects is copied in a contiguous fashion into the tospace; it is handled as if it were a queue. Consider now the first of these objects, and add to the end of the queue (that is, copy into the tospace) the objects pointed to by the pointers present in the first object, while, at the same time, these pointers are modified. In this way, we have copied into the tospace all the first object's children. We keep processing the queue until it remains empty. At this point, in the tospace we have a copy of the live objects in the fromspace.[20]

[20]Some precautions must be taken to prevent objects accessible via multiple pointers being copied more than once.

A stop and copy garbage collector can be made arbitrarily efficient, provided that there is enough memory for the two semi-spaces. In fact, the time required for a stop and copy is proportional to the number of live objects present in the heap. It is not unreasonable to assume that this quantity will be approximately constant at any moment during the execution of the program. If we increase the memory for the two semi-spaces, we will decrease the frequency with which the collector is called and, therefore, the total cost of garbage collection.

8.12 Chapter Summary

This chapter has dealt with a crucial aspect in the definition of programming languages: the organisation of data in abstract structures called data types. The main points can be summarised as follows.

- *Definition of type* as a set of values and operations and the role of types in design, implementation and execution of programs.
- *Type systems* as the set of constructs and mechanisms that regulate and define the use of types in a programming language.
- The distinction between *dynamic* and *static* type checking.
- The concept of type-safe systems, that is safe with respect to types.
- The primary *scalar types*, some of which are *discrete* types.
- The primary *composite types*, among which we have discussed *records*, *variant records* and *unions*, *arrays* and *pointers* in detail. For each of these types, we have also presented the primary storage techniques.
- The concept of *type equivalence*, distinguishing between equivalence by name and structural equivalence.
- The concept of *compatibility* and the related concepts of coercion and conversion.
- The concept of *overloading*, when a single name denotes more than one object and static disambiguation.
- The concept of *universal polymorphism*, when a single name denotes an object that belongs to many different types, finally distinguishing between parametric and subtype polymorphism.
- *Type inference*, that is mechanisms that allow the type of a complex expression to be determined from the types of its elementary types.
- Techniques for runtime checking for *dangling references*: tombstones and locks and keys.
- Techniques for *garbage collection*, that is the automatic recovery of memory, briefly presenting collectors based on reference counters, mark and sweep, mark and compact and copy.

Types are the core of a programming language. Many languages have similar constructs for sequence control but are differentiated by their type systems. It is not possible to understand the essential aspects of other programming paradigms, such as object orientation, without a deep understanding of the questions addressed in this chapter.