Tag-Free Garbage Collection: Two Variations on a Theme

sk and dbtucker

November 27, 2000

BIBOP

In the garbage collectors we've seen so far, we tag each object with its type. We would like to get rid of these tags entirely, since they take up space in memory. One possibility is to create a separate sections of the heap for each different type, and only allocate objects in their sections:

cons	box	vector	...

We can then determine the type of an object solely by its location in memory. For example, we know that if an object is in the cons section, it must be a cons pair; therefore, we don't need to store tag as well. What if your program creates many cons pairs, but few boxes? You could create an additional section for cons pairs:

cons	box	vector	...	cons

However, in order to compute the type of an object, we may have to do a large number of address comparisons. The operating system provides a convienent way of assigning sections of memory: virtual memory pages. We'll use each page for a single type of data, and the GC can then traverse the data just by knowing what page it's examining.

Now allocation is potentially expensive. First, to allocate a cons pair, you have to go looking for the last cons page with empty space. If each cons page has a pointer to the next cons page, you may have to follow many pointers to

find free space. Second, how can you tell when you have reached the end of a page? Remember the technique we used previously: we made the last page in memory unwritable. When you came to the end of free memory and tried to allocate on the unwritable page, you got a page fault. We can use the same technique here. Before we allocate memory in some page, we change the next page in memory to be unwritable. After we have allocated, we then flip the next page back to writable. However, we have to do this toggling every time we allocate an object, which is expensive.

The Chez Scheme implementation uses a strategy that avoids this work during allocation. This system allocates objects into a nursery, and tags the objects with their types. Then, when the garbage collector moves objects out of the nursery, it performs the above tasks; namely, following pointers to find a page with free space, and toggling the writability of memory pages. Since the GC is likely copying many objects out of the nursery, the costs associated with these tasks is amortized over the total number of copied objects during collection.

This method of garbage collection is called BIBOP (Big Ol' Bag of Pages).

GC in typed languages

Another approach to tagless garbage collection is to exploit type information. In a statically typed language, the run-time system only needs tags for datatype variants in order to ensure type safety. However, the GC requires that every object have a tag, so that it knows how to collect the objects. Can we modify our GC such that it only needs tags for variants?

When we traverse the heap in DFS, we can think of the heap as a big datatype:

```
(datatype Cell
  [Immediate (v value)]
  [Cons (car Cell)
        (cdr Cell)]
  [Box (v Cell)]
  [Vector (n number)
          (v1 Cell)
          ...
          (vn Cell)])
```

What is a garbage collector? It's a function that consumes a heap and performs some action on it, such as marking or copying:

```
(define (gc cell)
  (cases cell
    [Immediate (val) ...]
    [Cons (car cdr) (gc car) (gc cdr)]
    [Box (val) (gc val)]
    [Vector (v1 ... vn) ...]))
```

The traversal that a GC performs is the same kind of traversal as our interpreter did. You can think of the GC as an *interpreter* of the heap. The heap is then a program of instructions for the GC.

Since the GC is an interpreter, we know how to mechanically transform it into a compiler. What does compilation buy us over interpretation? We don't have to keep examining the input at run-time—in an interpreter, a source of inefficiency is that you do the instruction decoding phase over and over. The **cases** statement above is exactly this decoder. We can apply the idea of compilation to garbage collection, and get rid of the decoding phase.

We'll divide our GC into a number of small custom GCs, one for each type of value on the heap. Let's say we don't have polymorphism, and the values in our language are *num*s, *nlist*s (with two variants, *nempty* and *ncons*), and *nbox*es.

```
(define (gc ptr heap)
  (case (vector-ref heap ptr)
    [(Num) (gc/number ptr heap)]
    [(NEmpty) (gc/nempty ptr heap)]
    [(NCons) (gc/ncons ptr heap)]
    [(NBox) (gc/nbox ptr heap)]))

(define (gc/number ptr heap)
  'do-nothing)

(define (gc/nempty ptr heap)
  'do-nothing)

(define (gc/ncons ptr heap)
  (gc (+ ptr 1) heap)  ;; collect the car
  (gc (+ ptr 2) heap)) ;; collect the cdr

(define (gc/nbox ptr heap)
  (gc (+ ptr 1) heap)) ;; collect the contents
```

So far we have gained nothing. Now we can use our knowledge of types when garbage collecting components of data structures. For example, we know that an *nbox* always contains a *num*, so *gc/nbox* can invoke *gc/number* directly:

```
(define (gc/nbox ptr heap)
  (gc/number (+ ptr 1) heap)) ;; collect the contents
```

Similarly, we know that an *ncons* comprises a *num* and an *nlist*:

```
(define (gc/ncons ptr heap)
  (gc/number (+ ptr 1) heap)  ;; collect the car
  (gc/nlist (+ ptr 2) heap))  ;; collect the cdr

(define (gc/nlist ptr heap)
  (case (vector-ref heap ptr)
```

[(NEmpty) (*gc/nempty ptr heap*)]
[(NCons) (*gc/ncons ptr heap*)]))

There are two important points to note. First, we still must check tags for datatype variants, which we do in *gc/nlist*. Second, the only time we call the function *gc* is at the start—that is, for references in the root set. For all values on the heap (other than variants), we do not need to store their tags!

Since we've done away with tags, we can no longer tell what the structure of a stack frame is—which things are pointers and which aren't. So, we create a custom garbage collector for each variety of stack frame. Since stack frames are created by functions, each function needs a custom traversal (which invokes the *gc/type* function for each local variable). This custom collector needs to be stored somewhere; a good place is at some known offset from the return address.

Moral: Types are useful for garbage collection.

We have seen two ways of performing garbage collection without tags. Another, very different, way to accomplish tagless GC is to use conservative collection, the topic of the next lecture.