Fast Ephemeron Collection

Mark S. Miller
Google, Inc.

Andreas Gal
Mozilla Labs

Abstract

As far as we are aware, all previous published or implemented ephemeron collection algorithms have an $O(N^2)$ complexity measure during an atomic step of their marking phase. To ensure efficient collection under novel usage patterns, we first present an ephemeron collection algorithm with a reasonable complexity measure and reasonable constant overheads. We then present an elaboration with $O(1)$ complexity during each atomic step of the marking phase, where these steps can be interleaved with the mutator, allowing the mutator to remain responsive.

1 Introduction

Programmers often use weak-keyed hash tables to implement *soft fields*. A soft field acts, for most purposes, like a new field added to a set of objects—it associates each object in the set with a corresponding value for that field. However, rather than store these field values on the objects, instead it stores them in a side table indexed by the identity of these objects.

Were this side table a normal identity-keyed hashtable, soft fields would have an obvious memory leak compared to normal fields. When programmers use weak-keyed hashtables to avoid this obvious leak, they continue to suffer from a subtler leak that they are generally unaware of.[1] A weak-keyed hash table is implemented by a hash table storing weak references to its keys and strong pointers to its values. It also registers for notification of finalization on these weak key references (whether pre- or post-mortem). When notified, it deletes the association indexed by this key and thus drops its strong pointer to the corresponding value. The problem arises when the key is itself strongly reachable from the value. Since the table strongly retains the value and the value strongly retains the key, the collector cannot realize it can drop the key and so the table is never notified.

To solve the problem, the collector must know about the associations or the tables containing them directly, so it can mark the value *only* if the key *and* the table are already marked. When the collector directly knows about these

[1] I have informally observed this by quizzing several programmers, including garbage collection experts, of leak hazards they know of when using weak-keyed hash tables for soft fields.

associations, the associations are called *ephemeron pairs*. When the collector directly knows about the tables, the tables are generally called *ephemeron tables*. Either ephemeron pairs or ephemeron tables can be built from the other. In this paper, we consider only ephemeron tables. In the remainder of this paper, to distinguish levels of abstraction, we will adopt the term "*WeakMap*"[2] to refer to the application visible ephemeron table abstraction, reserving "*table*" for hash tables within the implementation. We define *ephemeron reachability* as the rule that the only values reachable from WeakMaps are those associated with reachable keys in reachable WeakMaps.

At any moment, say K is the set of all objects that are keys in any WeakMap, G is the set of WeakMaps, and KG the set of associations in all WeakMaps from these keys to values. We can divide each of these into two subsets according to whether they've been marked: K_u for the unmarked keys, K_m for the marked keys, etc. We can thus divide the associations into four disjoint subsets: $KG = K_uG_u \cup K_mG_u \cup K_uG_m \cup K_mG_m$. An ephemeron collector need only propagate markings to the values associated with K_mG_m.

The marking phase of previously published algorithms (**citations needed) and implementations (**citations needed) work essentially as follows:

```
Declare that all objects are white.
Color all roots grey.

While any objects are grey {
 While any objects are grey, pick a grey object x {
  If x is the get() function of a weak map {
   Note this weak map.
  } else {
   For all y's directly reachable from x {
     If y is white, color it grey.
   }
  }
  Color x black.
 }

 For each noted weak map g {
  For each k,v pair in g {  // rescanning for marked keys
   If k is black and v is white, color v grey
  }
 }
}

Retain black objects, recycling storage for white objects.
```

As with Dijstra's algorithm from which this is derived, white represents

[2] This is the agreed term for ephemeron tables as adopted by the next EcmaScript standard (**cite).

unmarked and black and grey represent marked. Grey further represents marked nodes from which potentially unmarked nodes might be reachable and need to be marked, so that the marking phase of the garbage collector is not done until all the grey is gone.

This rescanning potentially costs $O((|K_uG_m| + |K_mG_m|)^2)$. Consider the case where each value marking propagates to mark only one previously unmarked key. Ideally, we'd like our costs to be $O(|K_mG_m|)$, since those are precisely the associations that propagate to marking their associated values. The algorithm presented in this paper instead costs $O(|K_mG_u|+|K_uG_m|+|K_mG_m|)$. It wastes linear effort on the half-marked associations but wastes none on unmarked associations. We build up to this algorithm by successive refinement, starting with a simpler algorithm that does not perform as well.

2 A Non-Redundant Representation

We assume an internal hash table implementation with the following API, where `table.keys()` costs $O(|K|)$ cost and all other methods cost $O(1)$:

```
class Table {
   get(Object) -> Value
   set(Object, Value)
   keys() -> Object[]  // snapshot
   pick() -> Object | null
   reset()
 }
```

`Value` represents the type that includes all first class values in our language, including an assumed `null`. `Object` represents the subtype of `Value` including only those values with an unforgeable identity, because synthesizable values, such as the number 33, cannot usefully serve as keys in weak maps. We assume `null` is not an `Object`.

This provides a table as a total mapping from all objects to any value, where all objects map to null by default, and only a finite number map to any other value. Thus, `table.get(key)` always returns a valid value. With this representation, `table.set(key, null)` effectively deletes that key from the table. As a temporary measure to disappear in later refinements, tt table.keys() returns a list containing a snapshot of all keys that do not map to `null`. On an empty table, i.e., a table in which all keys map to `null`, `table.pick()` returns `null`. On a non-empty table, `table.pick()` returns some key that does not map to `null`. Finally, `table.reset()` deletes all associations from the table, so that all keys once again map to `null`.

In Figure 1, the four quadrants represent our four subsets

White, upper left corner represents K_uG_u, the set of all associations from unmarked keys in unmarked weak maps.

Yellow, upper right corner represents $K_m G_u$, the set of all associations from marked keys in unmarked weak maps.

Blue, lower left corner represents $K_u G_m$, the set of all associations from unmarked keys in marked weak maps.

Green, upper right corner represents $K_m G_u$, the set of all associations from marked keys in marked weak maps.

Each small circle represents an individual association, placed on a row according to its weak map identity and a columns according to its key identity. The oblong rounded rectangles represent our internal tables. The labeled ones outside the square are global tables whose existence is stable. These outer tables map from key identities to the inner tables shown hovering over the square, that map from weak map identities to values. Each such mapping is represented as an association shown over the inner table.

```
ByKey = [[Table(), Table()], [Table(), Table()]]

getValue(g :WeakMap, k :Object) -> Value {
  let outerTable = ByKey[isMarked(k)][isMarked(g)]
  let optInnerTable = outerTable.get(k)
  if (optInnerTable == null) { return null }
  return optInnerTable.get(g)
}
```

At the beginning of our marking phase, all objects are unmarked including our keys and weak maps, and so all associations are in the white upper left quadrant. Whenever a key gets marked, we need to move the column representing that key identity from the left to the right, represented by the rightward yellow arrows. Yellow added to white yields yellow. Yellow added to blue yields green. Whenever a weak map gets marked, we need to move the row representing that weak map identity from top to bottom, represented by the downward pointing blue arrows. As an association lands in the green lower right quadrant, if its value is unmarked, we color it grey.

We could mark a key by moving the inner tables themselves rightward

```
markKey(key :Object) {
  let optInnerWhite = ByKeys[0][0].get(key)
  ByKeys[1][0].set(k, optInnerWhite)

  let optInnerBlue = ByKeys[0][1].get(k)
  ByKeys[1][1].set(k, optInnerBlue)

  if (optInnerBlue != null) {
    for g in optInnerBlue.keys() {
      mark(optInnerBlue.get(g))
```

```
    }
  }
}
```

With this choice, marking a key is fast, as we simply transfer at most two inner tables from being stored in the left outer tables to being stored in the right outer tables, in order to move the column. To mark K_m keys, our complexity measure is $O(|K_m| + |K_m G_m|)$, where the last term is the cost of coloring grey all values for newly green associations. Instead, we opt for a choice that, at this stage, has a worse complexity measure but better sets us up for our next refinements.

Our new `markKey` depends on two new helper functions:

`getInnerTable(outer, key)` gets, or makes if necessary, an inner table at key in the outer table.

`move(outerFrom, outerKey, innerKey, innerTo)` If there is an association at outerFrom.get(outerKey).get(innerKey) move it to innerTo.get(innerKey), while cleaning up the tables it was removed from. Returns the transfered value, or null if none.

```
 // O(1)
getInnerTable(outer :Table, key) -> Table {
  var optResult = outer.get(key)
  if (optResult == null) {
    optResult = Table()
    outer.set(key, optResult)
  }
  return optResult
}

 // O(1)
move(outerFrom, outerKey, innerKey, innerTo) -> Value {
  let optInnerFrom = outerFrom.get(outerKey)
  if (optInnerFrom == null) { return null }
  let value = optInnerFrom.get(innerKey)
  if (value == null) { return null }
  innerTo.set(innerKey, value)
  innerFrom.set(innerKey, null)
  if (innerFrom.pick() == null) { // empty
    outerFrom.set(outerKey, null)
  }
  return value
}
```

With these helpers, our new `markKey` remains simple.

```
markKey(key :Object) {
  let optInnerWhite = ByKeys[0][0].get(key)
  if (optInnerWhite != null) {
    innerYellow = getInnerTable(ByKeys[1][0], key)
    while (null != (g = optInnerWhite.pick())) {
      move(ByKeys[0][0], key, g, innerYellow)
    }
  }
  let optInnerBlue = ByKeys[0][1].get(key)
  if (optInnerBlue != null) {
    innerGreen = getInnerTable(ByKeys[1][1], key)
    while (null != (g = optInnerBlue.pick())) {
      value = move(ByKeys[0][1], key, g, innerGreen)
      mark(value)
    }
  }
}
```

This increases the complexity measure of marking a key to $O(|K_m G_u| + |K_m G_m|)$, which is still within our stated goals. The situation is not so happy for marking a weak map.

```
markWeakMap(g :WeakMap) {
  for k in ByKeys[0][0].keys() {
    let innerWhite = ByKeys[0][0].get(k)
    let value = innerWhite.get(g)
    if (value != null) {
      innerBlue = getInnerTable(ByKeys[0][1], k)
      move(ByKeys[0][0], k, g, innerBlue)
    }
  }
  for k in ByKeys[1][0].keys() {
    let innerYellow = ByKeys[1][0].get(k)
    let value = innerYellow.get(g)
    if (value != null) {
      innerGreen = getInnerTable(ByKeys[1][1], k)
      move(ByKeys[1][0], k, g, innerGreen)
      mark(value)
    }
  }
}
```

In this representation, we must enumerate all upper inner tables in order to test if they contain an association for this weak map, in order to move that association to the lower tables. To mark G_m weak maps, our complexity measure is $O(|G_m| \times (|K_u| + |K_m|))$. This is proportional to the total area of the rows

moved down. If these rows are sparse, this is prohibitively more expensive than just moving the associations in those rows.

Finally, we augment our collector's mark step to call `markKey` and/or `markWeakMap` in addition to its normal marking step `colorGrey`, which we assume is itself $O(1)$.

```
mark(obj) {
  if (isMarked(obj)) { return }
  colorGrey(obj)
  if (obj might be a key in any weak map) { markKey(obj) }
  if (obj is a WeakMap) { markWeakMap(obj) }
}
```

In order to quickly test whether an object might be a key in a weak map, we assume an additional bit per object for this purpose, that is set when an object is used as a key in a `weakMap.set(key, value)` operation. We leave the efficient clearing of this bit as an exercise for the reader. Note that a weak map can also be a key in weak maps, so there is no "`else`" guarding the last "`if`".

At the end of our marking phase, we collect all unreachable associations by reseting the global white, yellow, and blue tables. We begin the next marking phase by swapping the global green and white tables, thereby declaring all remaining associations to be white.

```
reset() {
  ByKeys[0][0].reset()
  ByKeys[0][1].reset()
  ByKeys[1][0].reset()
  swap(&ByKeys[0][0], &ByKeys[1][1])
}
```

3 A Symmetric Representation

To avoid this explosive amount of work, we opt for two way redundancy to give ourselves a symmetric representation, shown in Figure 2. Our original representation remains, in which keys are the primary index and weak maps are the secondary index. To this, we add four more global tables, shown as the outer vertical oblong rectangles in Figure 2, in which the weak maps are the primary index and the keys are the secondary. Each association is now represented by entries in two inner tables, the original representing this association's column and a new one representing this associations's row.

```
ByWeakMaps = [[Table(), Table()], [Table(), Table()]]
```

Figure 3 depicts how our new `markKey` algorithm moves column k rightward. To maintain the redundant representation, it must do the work of our earlier `markKey`, but it must also remove these associations from all the inner tables representing the rows on which these associations are found. For example, it must

remove b from the left w row and remove f from the left y row. And`markKey` must do this extra work in time proportional only to the number of associations present in the column, not on the total area of the column. Fortunately, we can now enumerate the column to find the rows that need updating.

```
markKey(key :Object) {
  let optInnerWhiteColumn = ByKeys[0][0].get(key)
  if (optInnerWhiteColumn != null) {
    innerYellowColumn = getInnerTable(ByKeys[1][0], key)
    while (null != (g = optInnerWhiteColumn.pick())) {
      move(ByKeys[0][0], key, g, innerYellowColumn)
      innerYellowRow = getInnerTable(ByWeakMaps[0][1], g)
      move(ByWeakMaps[0][0], g, key, innerYellowRow)
    }
  }
  let optInnerBlueColumn = ByKeys[0][1].get(key)
  if (optInnerBlueColumn != null) {
    innerGreenColumn = getInnerTable(ByKeys[1][1], key)
    while (null != (g = optInnerBlueColumn.pick())) {
      value = move(ByKeys[0][1], key, g, innerGreenColumn)
      innerGreenRow = getInnerTable(ByWeakMaps[1][1], g)
      move(ByWeakMaps[1][0], g, key, innerGreenRow)
      mark(value)
    }
  }
}

markWeakMap(g :WeakMap) {
  // the transpose of the markKey algorithm, except omitting
  // the mark(value) call at the end which would be redundant
}
```

Taking both `markKey` and `markWeakMap` into account, the above code repeats essentially the same algorithm four times, corresponding to our big yellow and blue arrows in Figure 1. An actual implementation might express this algorithm in a more parameterizable form, so that it can be reused in these four contexts. We avoided doing so in this paper for expository purposes.

4 Worst Case Complexity Measure

Notice that `markKey` and `markWeakMap` no longer make any use of `table.keys()`, which we can therefore now drop from our internal Table abstraction. All the remaining operations they call are each $O(1)$, so we can calculate the complexity according to how many times each loop body is executed. For example, the first loop of `markKey`, corresponding to the upper rightward yellow arrow in Figure 1, executes $|K_m G_u|$ times. Altogether, we have $|K_m G_u| + |K_m G_m| + |K_u G_m| +$

$|K_m G_m|$ loop iterations during the marking phase, achieving our original goal of $O(|K_m G_u| + |K_u G_m| + |K_m G_m|)$

5 Actual Implementation and Measurement

(**TBD*)

6 Incremental Collection

(**TBD*)

7 Usage for Membranes

(**TBD*)

8 Conclusions

(**TBD*)

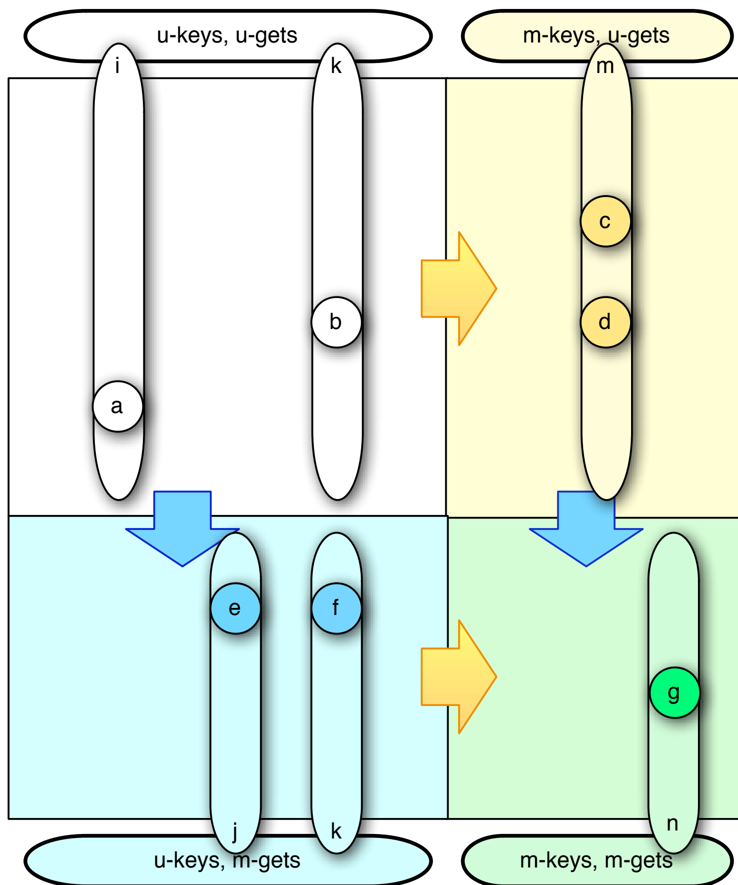

Figure 1: By-key Representation.

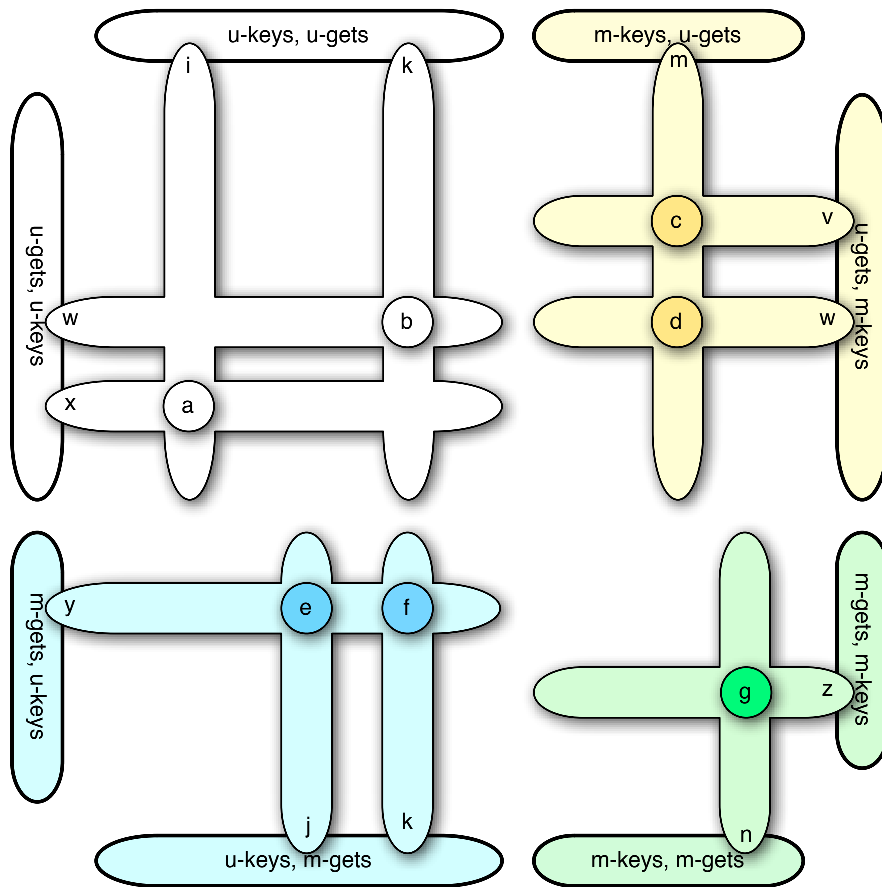

Figure 2: Symmetric Representation.

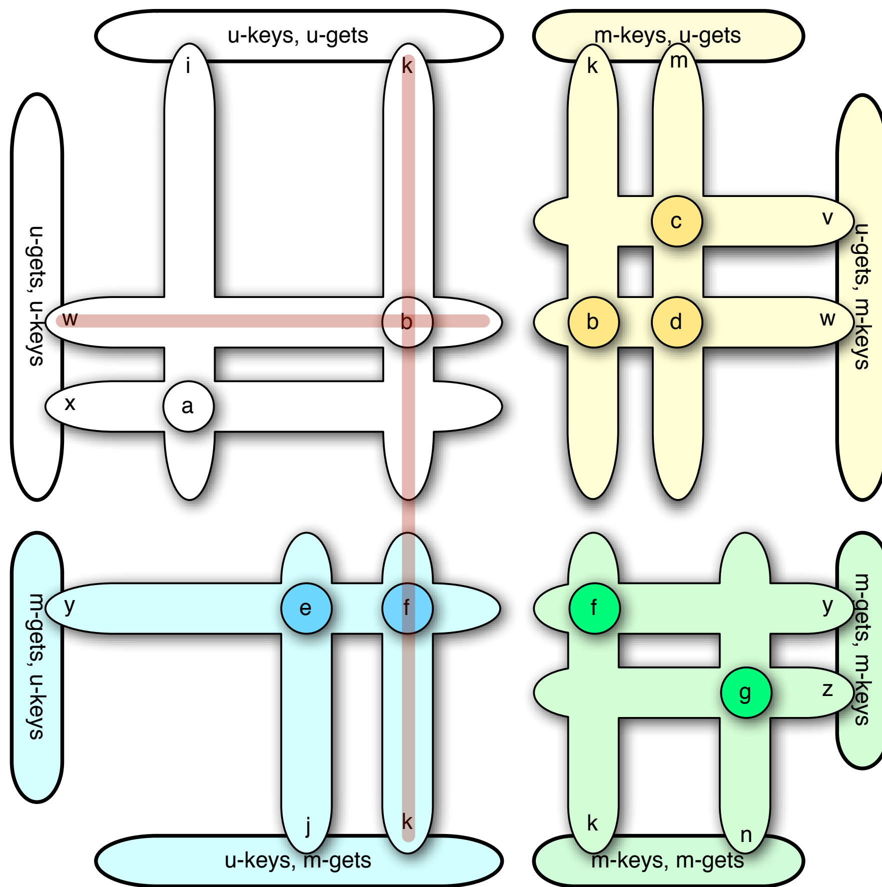

Figure 3: Marking a Key.