Dynamic Region Inference

David Pereira John Aycock

Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4
{pereira|aycock}@cpsc.ucalgary.ca

TR 2002–709–12

Abstract

We present a garbage collection scheme based on reference counting and region inference which, unlike the standard reference counting algorithm, handles cycles correctly. In our algorithm, the fundamental operations of region inference are performed dynamically. No assistance is required from the programmer or the compiler, making our algorithm particularly well-suited for use in dynamically-typed languages such as scripting languages. We provide a detailed algorithm and demonstrate how it can be implemented efficiently.

1 Introduction

Reference counting is a garbage collection strategy which provides immediate reclamation of unreachable objects and distributes garbage collection delays over the entire execution of the program. As such, it is an attractive method of garbage collection, especially for scripting languages and interactive applications.

The basic idea of reference counting is to keep a count of the number of references to each object; when an object's reference count becomes zero, it is not referenced by any other object and can be reclaimed. To do this, an object's reference count is incremented when a reference to it is created. Likewise, when a reference to an object is deleted, the object's reference count is decremented. However, this method has a fatal flaw – the inability to collect cycles of objects [11]. Figure 1 shows the creation of an unreachable, uncollectable cycle.

Our approach to reference counting is based on the concept of *regions*. A region is a subset of the set of objects of a running program. Regions are usually defined explicitly by the programmer or inferred automatically by the compiler. Most importantly, a region is treated as a single unit by the garbage collector;

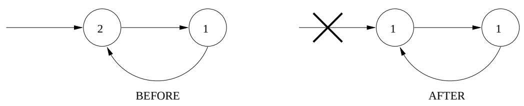

Figure 1: Creation of an isolated cycle. Objects are depicted as circles surrounding their reference counts.

vital statistics, such as the reference count, are associated with the entire region, instead of with each object within it. The objects in a region are also usually deallocated *en masse*.

In this paper, we present a new region-based reference counting garbage collection method. Our algorithm differs from other region-based approaches since it infers regions at run-time, and is designed primarily for dynamically-typed languages such as Smalltalk and Python, both of which have traditionally used reference counting systems, and whose type systems do not make them amenable to the methods of static region inference. Our method also handles cycles correctly.

2 Related Work

Many attempts have been made to remedy the failure of cyclic reference counting, both specific and general. Friedman and Wise [6] discovered a technique specific to functional languages. Bobrow [2] created a technique wherein objects are assigned to groups by the programmer, and a reference count is kept for the entire group. By distinguishing internal references – pointers from an object in a group to other objects in the *same* group – from external references, intra-group cycles can be handled; internal references do not contribute to the region reference count. When the group reference count drops to zero, the entire group can be reclaimed. However, this method does not handle inter-group cycles, and requires programmer intervention to '…declare in advance some information concerning the use of pointers in cells of the structure.' [2, page 270]

Other techniques are more general. They can be divided into three major categories: weak-pointer algorithms, partial mark-sweep algorithms, and static region inference.

Weak-pointer algorithms differentiate cycle-closing pointers from non-cycle-closing pointers. The most famous weak-pointer algorithm is that of Brownbridge [3]. A fatal error in the algorithm, the premature reclamation of objects, was corrected by Salkild [15]. Pepels [12] then corrected the Brownbridge-Salkild algorithm, which is not guaranteed to terminate, and proved it correct. However, Pepels' algorithm is complicated, and its time complexity is exponential in the worst case.

Christopher [4] designed a simple partial mark-sweep algorithm wherein a mark-sweep collector is invoked to reclaim the cycles missed by reference counting. Lins [8] created a hybrid reference counting/mark-sweep algorithm which is $O(n^2)$ in the worst case, based on an algorithm of Martinez [10]. Bacon et al. [1] designed an $O(V+E)$ variant of the Lin's algorithm for the IBM Jalapeño Java Virtual Machine. This was further improved upon by Lins in [9], which contains an efficient general reference-counting algorithm. In contrast to our algorithm, these algorithms require varying amounts of mark-sweep collection to be performed as part of their core functionality.

Static region inference has been used with great success in functional languages such as ML [16, 17, 18]. These techniques have also been applied to non-functional languages such as C [7].

3 Algorithm

We consider the object set of a program to be comprised of three distinct types of object:

1. Aggregate objects, which can contain references to other objects. The lists, tuples, and dictionaries of most scripting languages fit into this category.

2. Non-aggregate objects, which cannot refer to other objects. Integer and string objects would be non-aggregate objects, for instance.

3. Machine registers, which contain a single reference to an aggregate or non-aggregate object, but cannot themselves be referenced. As the name implies, these would map to registers in native code or a register-based virtual machine.

In our algorithm, a region is a collection of aggregate objects. Furthermore, it is a maximal collection of objects; any aggregate object reachable from an object in a given region is a member of that region.

Initially, each aggregate object is trivially a region. Larger regions are built when inter-region references are created; if the two aggregate objects involved are not in the same region, then the regions that they belong to are merged into one. This is one aspect of dynamic region inference, which we call *region merging*. It is important to note that, given our method of region merging, the creation of a reference which will result in the creation of a cycle will be an intra-region reference, which is not counted (see Section 3.2.1 below). As a result, the non-zero reference counts which insulate each member of a cycle from collection in classical reference counting algorithms will not be produced.

The opposite operation is *region splitting* – it is conceivable that references may be deleted so that a region degenerates into two or more unconnected components. In this case, region splitting is performed to break that region into two or more new regions. Together, region merging and region splitting form our dynamic region inference algorithm.

3.1 Requirements

Since this algorithm makes a distinction between machine registers, aggregate and non-aggregate objects, it is required that objects can be distinguished as such. In a statically typed language, this is done at compile time. In a dynamically typed language, objects must be annotated with enough information so that this distinction can be made at run-time. In the following code, we shall assume the dynamic case (for the purposes of exposition) and use the predicate `CategoryOf()` to make the distinction.

A Region Information Table (RIT) is used to store the vital statistics of each region in the program; each RIT entry corresponds to one region. A reference count field, `RC`, is kept for each region in the RIT. In addition, each RIT entry contains a pointer, `MEMBERS`, to the "first" member of the region.

Each object requires a set of ancillary data fields for use by the algorithm. A singly-linked list is threaded though all the members of a region using a `NEXT` field in each aggregate object, and it is thus possible to find all the members of a given region. Each region member contains a "back-link" to the appropriate entry in the RIT, called `REGION`. Non-aggregate objects do not belong to any region, and each non-aggregate object carries its own reference count.

Note that any two aggregate objects in the same region will have the same value in their `REGION` fields. This invariant is used to test whether two aggregate objects are members of the same region.

Finally, let `UPDATE(source, destination)` represent the creation of a reference from object `source` to object `destination`, where `source` and `destination` are not necessarily distinct. Also, let `DELETE(source, destination)` represent the deletion of a reference between the same. These procedures are the *mutators* for the object graph; only they can change the reachability of objects from the *root set*, a set of predetermined objects. Objects that cannot be reached from the root set are deemed to be garbage.

3.2 Mutator Operations

In this section, we describe the functionality which the mutators `UPDATE` and `DELETE` implement. We first give a conceptual, pseudocode version of these algorithms; implementation details follow.

3.2.1 Reference Creation

By definition, `UPDATE` requires `source` to be either a register or an aggregate object, and `destination` to be either an aggregate or non-aggregate object.

The most important case is when both `source` and `destination` are aggregate objects. In this case, a check is made to ascertain if both objects are in the same region. If they are in different regions, the two regions are merged, as shown in Figure 2. There are two alternatives here: should the region containing `source` should be merged into the region containing `destination` or vice-versa? In the implementation section it will be shown how the efficiency

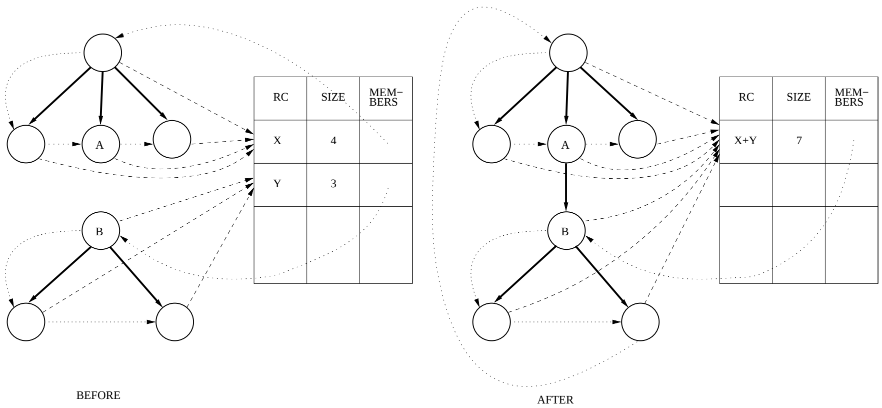

Figure 2: Objects A and B are in different regions. The creation of a reference from A to B results in region merging. Solid arrows are references, dashed arrows are back-links, and dotted arrows are links in the list threaded through objects in a region.

of merging can be improved by making this choice properly. For now, we shall call the region into which members are added the *parent region* and the other region the *child region*. The merging process consists of the following steps:

1. The reference count of the child region is added to the reference count of the parent region. Since the parent region and the child region are two different regions, they are disjoint and the reference count of the parent region after merging is equal to the sum of the reference counts of the parent region and child region prior to merging.

2. All `REGION` fields in the members of the child region are changed eagerly to point to the RIT entry of the parent region.

3. The list of members of the child region is appended to the list of members of the parent region.

4. The RIT entry for the child region is recycled, since it is no longer required.

If `source` is a register and `destination` is an aggregate object, the reference count of the region containing `destination` is incremented. If `source` is a register or aggregate object and `destination` is a non-aggregate, normal reference counting is performed. Pseudocode for reference creation and region merging is shown in Figure 3.

3.2.2 Reference Deletion

`DELETE`, by definition, requires `source` to be either a register or an aggregate object, and `destination` to be either an aggregate or non-aggregate object. Its pseudocode is shown in Figure 4.

```
UPDATE(source, destination):
    select (CategoryOf(source), CategoryOf(destination)):

    case (Aggregate, Aggregate):
        if source.REGION <> destination.REGION
            parentRegion = source.REGION
            childRegion  = destination.REGION
            MergeRegions(parentRegion, childRegion)

    case (Register, Aggregate):
        increment destination.REGION.RC

    case (Register,  Non-Aggregate):
    case (Aggregate, Non-Aggregate):
        increment destination.RC

MergeRegions(parentRegion, childRegion):
    parentRegion.RC += childRegion.RC
    for each member in childRegion:
        member.REGION = parentRegion
    append childRegion.MEMBERS to parentRegion.MEMBERS
    ReclaimRITEntry(childRegion)
```

Figure 3: Pseudocode for reference creation and region merging. `ReclaimRITEntry` is a function which places the given RIT entry on a free list of table entries.

```
DELETE(source, destination):
    select (CategoryOf(source), CategoryOf(destination))

    case (Aggregate, Aggregate):
        /* No operations required. */

    case (Register, Aggregate):
        decrement destination.REGION.RC
        if destination.REGION.RC = 0
            ReclaimRegion(destination)

    case (Register,  Non-Aggregate):
    case (Aggregate, Non-Aggregate):
        decrement destination.RC
        if destination.RC = 0
            ReclaimObject(destination)
```

Figure 4: Pseudocode for reference deletion. In the code above, `ReclaimRegion` is a function which deallocates all the objects in a region, and then places the RIT entry for that region onto the free list. `ReclaimObject` is a function which deallocates a single non-aggregate object.

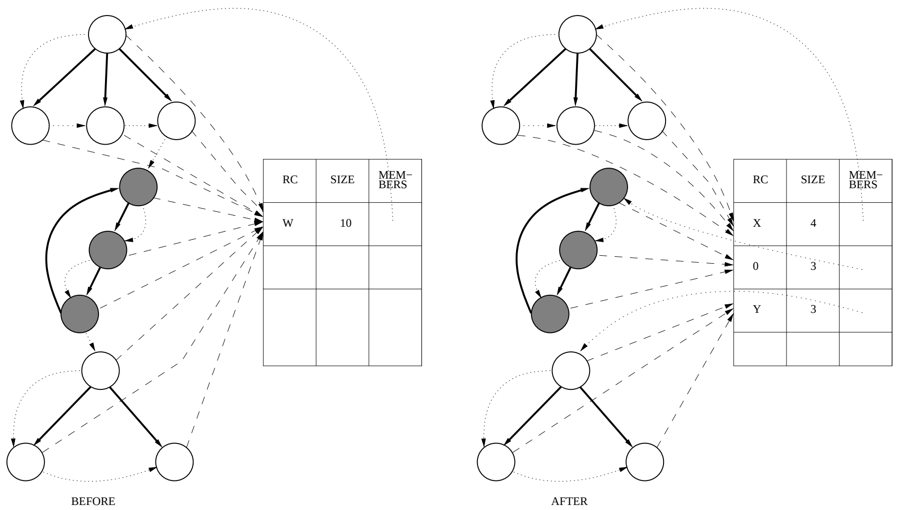

Figure 5: Region splitting. Here, the region composed of the shaded objects has a reference count of 0 after splitting, and the memory occupied by the objects in it can be reclaimed. (In practice, an entry would not have been made for it in the RIT.)

Again, the most important case is when both `source` and `destination` are aggregate objects. In this case, both source object and destination objects belong to the same region; the initial creation of the reference between them would have merged both objects into the same region if they were not already. Hence, an internal pointer is being deleted and no further operations are required. If `source` is a register or aggregate object and `destination` is a non-aggregate, normal reference counting is performed. Otherwise, if `source` is a register and `destination` is an aggregate object, the reference count of the region containing `destination` is decremented. If the reference count of the region becomes zero as a result of this operation, the entire region can be reclaimed.

4 Region Splitting

The deletion of intra-region references may cause a region to degenerate into two or more unconnected components. Naturally, if one or more components in a region are not reachable, we would like to deallocate those components. However, since objects in those components are members of the region, they cannot not be deallocated until the entire region is reclaimed. This can potentially delay the reclamation of a program's unreachable data. Region splitting, illustrated in Figure 5, is used to alleviate this.

Region splitting is an incremental procedure; a *single* candidate region is selected from the Region Information Table and then split, if possible. Region splitting need not be done until absolutely necessary; it can be triggered by the memory allocator. For example, when the amount of memory allocated by a program exceeds a certain predefined threshold, region splitting can be performed to reclaim the memory occupied by unreachable objects.

To split a region, its components are computed. That is, the region is partitioned into equivalence classes where two objects are in the same class if and only if one is reachable from the other. A new region is then created from each component. Since the original region contained the *total* reference count of all external references to its members, the reference counts for individual members are not known and must be calculated. We can calculate this without following all pointers in the heap because the register set forms the root set of all the objects in any given region. Hence, by traversing the register set, the reference count for each new region can be calculated. Our approach is shown in Figure 6, where the notation `[reg]` is used to denote register dereferencing.

5 Implementation

The implementation builds upon the conceptual algorithm in two ways.

The first modification optimizes the merging of regions. When two regions are required to be merged into one, the `REGION` field of each member of the child region needs to be set to point to the RIT entry of the parent region. This requires the region member list of the child region to be traversed. It is thus desirable for the child to be the region with fewer objects. To implement this, a field, `SIZE`, containing the number of members in a given region is kept in each RIT entry.

The second modification is used to optimize the check to ascertain whether two objects are in the same region. Conceptually, both `UPDATE` and `DELETE` contain tests to determine the categories of the source and destination objects. If they are both aggregate objects, another test is made to ascertain whether they are members of the same region. This sequence of two tests can be folded into one. A key observation is that the usage of machine registers are always identifiable (to a code generator for a statically-typed language, or to virtual machine instructions in a dynamically-typed language) so that we can split the code for `UDPATE` and `DELETE` into two variants - one where `source` is always a register, and another where `source` is never a register (i.e., `source` is an aggregate object). The latter variant can be further optimized.

In the aggregate-as-source variant, there are two possibilities for `source` and `destination` objects: aggregate/aggregate and aggregate/non-aggregate. Two tests still seem to be required. If, however, a dummy `REGION` field is added to each non-aggregate object, and set to always point to its `RC` field, the check can be eliminated; given any two objects they are in the same region if and only if both `REGION` fields have the same value. One should note that the cost of this improvement is an extra field in every non-aggregate object – we exchange space efficiency in the layout of data structures for a decrease in the amount of code which performs the `UPDATE` and `DELETE` operations.

```
SplitRegion(region):
    componentSet = CalculateComponents(region)

    /* If the region can be split */
    if |componentSet| > 1
        /* Create a new region from each component */
        newRegionSet = ∅
        for each component in componentSet
            newRegionSet =
                newRegionSet ∪ {BuildNewRegion(component)}

        /* Calculate the total reference count for each region */
        for each register
            if CategoryOf([register]) = Aggregate
                if [register].REGION ∈ newRegionSet
                    increment [register].REGION.RC

        /* Free any unreferenced regions */
        for each region in newRegionSet
            if region.RC = 0
                ReclaimRegion(region)

BuildNewRegion(component):
    newregion = AllocateRITEntry()
    newregion.MEMBERS = Λ
    newregion.RC = 0
    for each object in component:
        object.REGION = newregion
        append object to newregion.MEMBERS
    return newregion
```

Figure 6: Pseudocode for region splitting. `CalculateComponents` partitions the graph of objects in a given region into its components. It employs a union-find algorithm, a well-known class of algorithm to do this [5]. The empty list is denoted by Λ.

5.1 Region Information Table

The region information table is an array of fixed length records, each with the following structure:

Field Name	Size	Description
`RC`	Unsigned Integer	The number of external references to member objects of the region.
`SIZE`	Unsigned Integer	The number of member objects in the region.
`MEMBERS`	Machine Pointer	The first item in the Region Member List.

A free list is threaded through the RIT to facilitate rapid allocation and deallocation of entries. It should be noted that the `SIZE` field does *not* have to be large enough to represent the number of objects in the region, since no error will occur if the parent and child are chosen incorrectly. For this purpose, a smaller field with a saturating count can be used.

5.2 Object Layout

5.2.1 Aggregate Objects

Field Name	Size	Description
`REGION`	Machine Pointer	Points to the containing region's RIT entry.
`NEXT`	Machine Pointer	Points to the next object in the region.

Recall that `RC` is the first field of an RIT entry. Hence, the `REGION` field of an aggregate object points directly to `RC`, and no offset calculations are required to access this commonly-used field.

5.2.2 Non-Aggregate Objects

Field Name	Size	Description
`REGION`	Machine Pointer	Points to the `RC` field.
`RC`	Unsigned Integer	Object reference count.

In addition to optimizing the check to ascertain whether two objects are in the same region, the `REGION` field in non-aggregate objects has three other uses:

1. Since the (dummy) `REGION` field points to the `RC` field in non-aggregates, non-aggregate objects can easily be distinguished from aggregate objects, whose `REGION` fields will point into the RIT.

2. The use of the `REGION` field facilitates factoring the reference count information for a non-aggregate object into an auxiliary table. This centralizes all reference counts to a single location in memory, so that objects will not be dirtied when their reference counts change.

3. Since the first field of a RIT entry is the reference count field, the `REGION` fields of both aggregate and non-aggregate objects point directly to reference counts, thereby enabling the reference count of a non-aggregate object or the reference count of an entire region (given an aggregate object) to be accessed and modified with the same code. For example, to increment the reference count of object `O`, the code `increment O.REGION.RC` works regardless of whether `O` is an aggregate or non-aggregate object.

5.3 Region Splitting Implementation

To split a region into its components, a union-find algorithm with union-by-rank and path-compression[5] is used. This requires two extra fields in each node – a pointer to a parent, and an integer rank. The `REGION` and `NEXT` fields can be re-used to store these ancillary data. Since all members in the region have the same value for the `REGION` field, this field's contents need not be preserved during splitting. However, each object's `NEXT` field has a different value. The address of each object in the region can be temporarily placed into an array during splitting, and objects' `NEXT` field can be re-used.

6 Conclusion

We have presented a reference counting garbage collection algorithm based on dynamic inference of regions, which, unlike other algorithms, does not require any assistance from the compiler or the programmer, and handles cycles correctly. Our algorithm is especially well-suited for use in dynamically-typed languages such as scripting languages, which often employ reference counting garbage collection schemes.

References

[1] David Bacon, Dick Attanasio, Han Lee, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In PLDI [13].

[2] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[3] David R. Brownbridge. Cyclic reference counting for combinator machines. In Jean-Pierre Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, Nancy, France, September 1985. Springer-Verlag.

[4] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[6] Daniel P. Friedman and David S. Wise. Reference counting can manage the circular environments of mutual recursion. *Information Processing Letters*, 8(1):41–45, January 1979.

[7] David Gay and Alex Aiken. Language support for regions. In PLDI [13].

[8] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

[9] Rafael D. Lins. An efficient algorithm for cyclic reference counting. *Information Processing Letters*, 83:145–150, 2002.

[10] A. D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[11] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[12] E. J. H. Pepels, M. C. J. D. van Eekelen, and M. J. Plasmeijer. A cyclic reference counting algorithm and its proof. Technical Report 88–10, Computing Science Department, University of Nijmegen, 1988.

[13] *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[14] *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[15] Jon D. Salkild. Implementation and analysis of two reference counting algorithms. Master's thesis, University College, London, 1987.

[16] Mads Tofte. A brief introduction to Regions. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 186–195, Vancouver, October 1998. ACM Press.

[17] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998.

[18] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997. An earlier version of this was presented at [14].