

Container Types for Automatic Garbage Collection in Hard Real-Time Computing

Kevin Cleereman

Air Force Research Laboratory, Collaborative Technology Branch (IFSD)
2241 Avionics Circle, Wright-Patterson AFB, OH 45433-7334

Abstract – We present a container type system that will permit the use of reference counting on circular memory structures. Reference counting can be interleaved with program execution; therefore these container types present the potential of enabling automatic heap management in hard real-time computing applications.

Keywords: Garbage collection, reference counting, type systems, real-time computing

1.0 Introduction

It is extremely difficult to achieve type safety in a programming language that permits manual heap management. If memory is freed too soon, then dangling pointers may access memory that has been re-allocated. For this reason, modern programming languages like C# and Java use garbage collectors to manage the heap. Unfortunately, automatic garbage collection is inappropriate for hard real-time computing. Most garbage collector implementations cannot be preempted, and those that can be preempted may starve.

“To date, the [Real-Time for Java] expert group believes that no garbage collector algorithm or implementation is known that allows preemption at points that leave the inter-object pointers in the heap in a consistent state and are sufficiently close in time to minimize the overhead added to task switch latencies to a sufficiently small enough value which could be considered appropriate for all real-time systems.” [1]

Reference counting is generally less efficient than mark/sweep or copy collectors, but offers the advantage of incrementally freeing memory. This raises the possibility of using reference counting to interleave program execution and memory management, permitting the use of automatic garbage collection in hard real-time computing. However, reference counting fails in the face of circular memory structures [2], such as doubly-linked lists and circular lists. A garbage collector using reference counts may be paired with an auxiliary garbage collector that will free circular memory structures [3], but then we lose the advantage of incremental de-allocation and we can no longer make hard real-time guarantees about garbage collection. Alternatives require the programmer to explicitly de-allocate circular memory structures by deleting pointers from a cyclic data structure to break the loop, or to mark certain pointers as “grouped” [4] so that intra-group cycles can be deallocated. Such explicit intervention is error-prone – the programmer may de-allocate a circular structure too soon (resulting in invalid objects or dangling pointers) or not at all (resulting in a memory leak), or the programmer may introduce an inter-group cycle, thus re-introducing the problems that automatic memory management is intended to solve. Cyclic

memory structures can be reference counted with a runtime heap trace [5], but it can be just as difficult to make hard real-time guarantees on runtime cycle detection as on a mark/sweep collector.

We present a system of container types to solve the reference count problem with circular memory structures. We contend that the type system imposes minimal restrictions on the programmer, and that type checking is tractable. Our intention is to develop a type-safe language that is appropriate for hard real-time computing.

2.0 Recursive Types

A non-recursive type can be appropriately de-allocated through reference counting (Figure 1) – when the reference count to `Simple` decrements to 0, then `Simple` can decrement the reference counts to `x`, `y`, and `z`. However, recursive types cannot be guaranteed to form non-circular structures without performing expensive run-time type checks. For example, the recursive types in (Figure 2, Figure 3) *may* form non-circular memory structures, but as demonstrated in the pointer diagrams they may form circular lists that cannot be de-allocated with reference counts. To detect one of these cycles would require an $O(n)$ run-time check every time an element is added to the data structure, or a $O(n + e)$ heap trace as in Bacon’s algorithm, resulting in a potentially unacceptable performance loss.

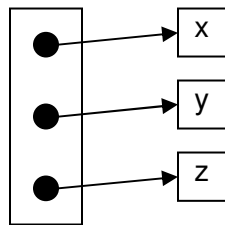
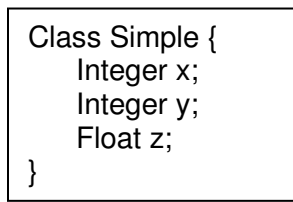


Figure 1: Non-recursive type.

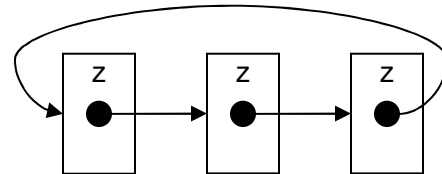
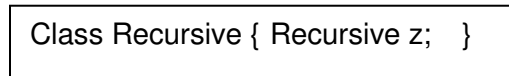


Figure 2: Recursive type.

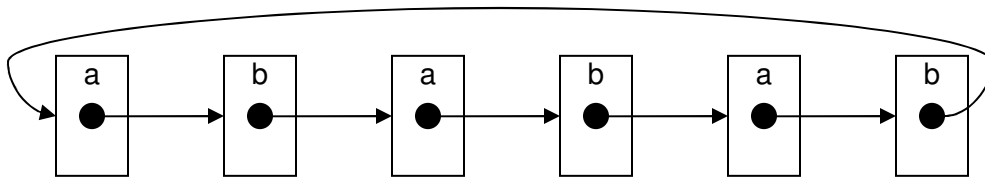
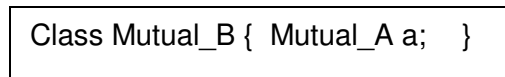
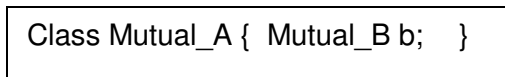


Figure 3: Mutually-recursive types.

Generic programming techniques, such as the C++ STL, can begin to provide a solution to the problem by providing pre-packaged circular containers. For example, in (Figure 4), we can use a `List` object with `head` and `tail` pointers to access a doubly-linked list of elements. The `front` and `back` pointers between the elements `{e1, e2, e3, e4}` are internal to the `List` object, so when the `List` reference count decrements to 0 its destructor can delete its internal pointers and allow the `{e1, e2, e3, e4}` reference counts to appropriately decrement.

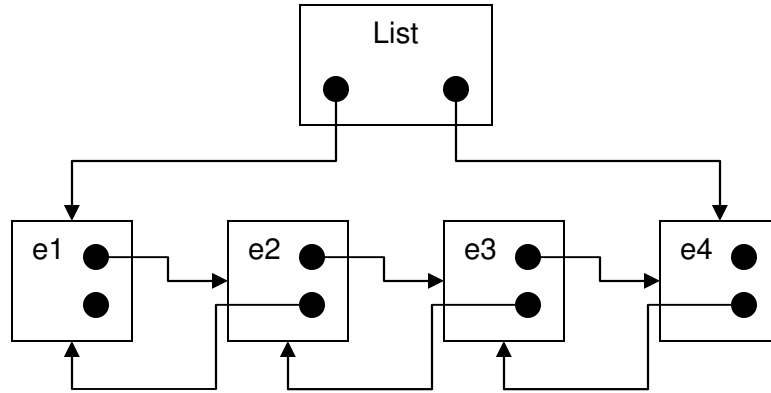


Figure 4: Generic List object with internal pointers.

Generic programming does not provide a full solution to the problem. We can use the STL to create containers with simple type properties, but we cannot create containers with arbitrary type properties. For example, in (Figure 5), we have made a binary tree with back pointers and two different node types {NodeA, NodeB}. It is not reasonable to expect a generic programming library to contain templates for such arbitrarily complex containers.

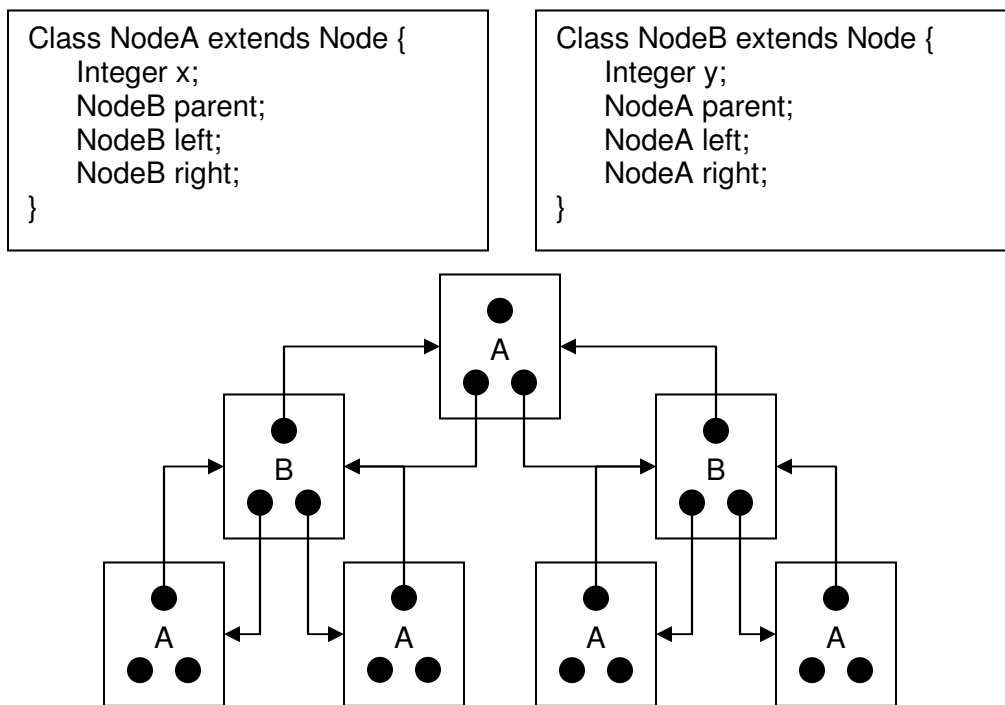


Figure 5: Binary tree with two node types. Pointers to Integer elements are not shown.

3.0 Container Types

Our solution to the problem of counting references for arbitrarily complex recursive types is to require the programmer to use container types. A container type is composed of two parts: the *free* class and the *bound* class. A type's free class holds the fields and methods that are bound directly to a variable within a

container, and a type's bound class holds the fields and methods that are bound to the container holding the variable. Neither class may contain recursive references; however the bound class is treated as a different type than the free class for purposes of determining whether references are recursive.

For example, we can rewrite our binary tree from (Figure 5) as follows:

```
Class NodeA extends Node {
    Integer x;
}

Class NodeB extends Node {
    Integer y;
}

Container BinTree {
    Node head;
}

Container BinTree:NodeA {
    NodeB parent;
    NodeB left;
    NodeB right;
}

Container BinTree:NodeB {
    NodeA parent;
    NodeA left;
    NodeA right;
}
```

The `Integer x` and `Integer y` variables are in the free class of `NodeA` and `NodeB`, respectively, meaning that they can be referenced even when `NodeA` or `NodeB` escapes its container. The `NodeA` and `NodeB` references have been moved from the class declaration to the container declaration, placing them in the bound class.

As stated, the free class members may be accessed independently of the container.

```
NodeA foo = new NodeA();
Integer bar = foo.x;
```

However, bound class members may only be accessed through a container. We present two alternative syntaxes for accomplishing this.

```
NodeA foo = new NodeA();
BinTree bar = new BinTree();
bar.head = foo;
NodeB temp1 = bar.foo.left;
NodeB temp2 = bar.foo.right;

BinTree bar = new BinTree();
Inside bar {
    NodeA foo = new NodeA();
    head = foo;
    NodeB temp1 = foo.left;
    NodeB temp2 = foo.right; }
```

The `Inside container { contained.bound_field }` notation is intended to enhance the readability and writability of the program. We would like to minimize the number of `container.contained.bound_field` notations that the programmer needs to make.

With all potentially recursive `NodeA` and `NodeB` references in the bound class of the `BinTree` container, we are now able to dereference all internal pointers to `NodeA` and `NodeB` objects when the `BinTree` variable goes out of scope (if it is stack allocated) or has its reference count decremented to 0 (if it is heap allocated).

4.0 Limitations

Reference counting is generally less efficient than mark/sweep or copy garbage collectors. Any serious use of reference counting for hard real-time computing will require an optimizing compiler to minimize the number of references being counted and to minimize the space taken up by the reference accumulators. We may be able to alleviate the cost of reference counting and possibly even improve on the performance of mark/sweep or copy collectors if we count references on regions rather than on objects [6], but it will not always be appropriate to treat a container type as a memory region, e.g., when elements escape their containers. It may be possible to determine through static analysis whether it will likely be more efficient to copy escaping elements to a new region to permit an entire container to be garbage collected in a single operation.

It is difficult to use container types for creating containers of arbitrary depth. For example, to create a list that contains lists that contains lists, the programmer will need to create three separate containers, e.g., List1 holding a list of List2's each holding a list of List3's (Figure 6), because the lists cannot contain free references to lists of the same type. This means that the programmer must know in advance the maximum depth of the containers in the program and create a separate container type for each level, or the programming language must allow some means of creating new container types at runtime, or the programming language must allow the use of dependent types in specifying containers.

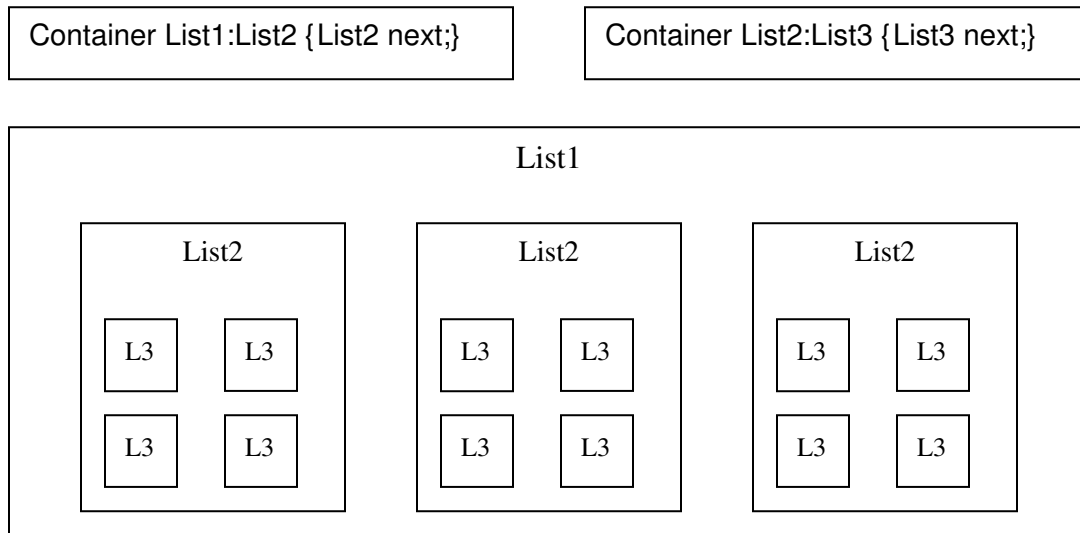


Figure 6: List1 contains List2's containing List3's.

A simpler solution to the problem of creating containers of arbitrary depth may be to use run-time checks to detect cycles in classes with a `Recursive` type.

```
Recursive Class List {  
    List data;  
    List next;  
}
```

The garbage collector can now use Bacon's algorithm to detect cycles in the `List` class, but instead of scanning the entire heap it need only scan the `Recursive` objects on the heap.

The issue of container inheritance has not yet been fully explored. A base type may have bound fields in multiple container types, e.g., `NodeA` may have bound fields (i.e., can be contained in) both a `Binary Tree` and a `Queue`. However, it may generate unnecessary complications if we allow `Binary Tree` to inherit from `Queue` (and thus allow `NodeA` to access the `Queue` fields from within a `Binary Tree`).

Finally, the issue of usability needs further investigation. We would like the programmer to be able to circumvent the `container.contained.bound_field` syntax wherever possible so that container types do not hinder program readability and writability, and for this purpose we have introduced the

```
Inside container { contained.bound_field }
```

syntax. However, we need to specify a way for the programmer to work inside of multiple containers at once, even given the fact that some containers may share bound field names. An aliasing scheme, e.g.,

```
Inside container1, container2 (alias1 container1.contained.bound_field,
    alias2 container2.contained.bound_field) { alias1 = alias2 }
```

may solve the problem, but this requires further investigation.

5.0 Complexity

A straightforward type-checking algorithm for confirming that there are no recursive types uses a topological sorting algorithm running in $O(n + e)$ time, where n denotes the number of nodes and e denotes the number of edges in the type graph (Figure 7). Every node represents a programmer-defined type, and every edge represents a reference within a programmer-defined type to another programmer-defined type. Note that we do not consider the references to primitive or library types (e.g., `Integer a` and `Float b` in `TypeY` and `TypeZ`, respectively) because we assume that these types cannot contain references to programmer-defined types and thus cannot form cycles in the type graph. If the language allows the programmer to redefine primitive and library types then we must do away with this assumption and consider *all* references in the cycle detection algorithm.

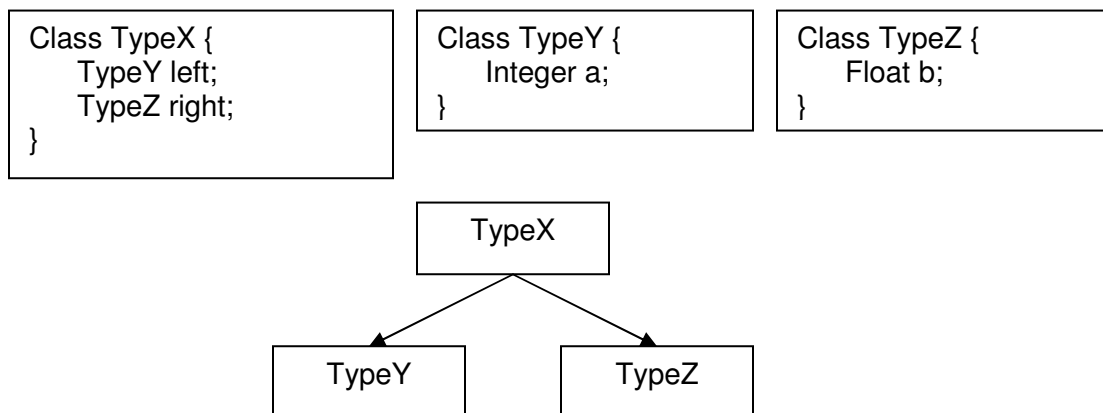


Figure 7: Three types and their resulting type graph.

The type-checker uses a topological sorting algorithm to verify that there are no cycles in the directed type graph. Fortunately, if we introduce dynamic types to allow the programmer to create containers of arbitrary depth, we may use a more efficient incremental topological sort algorithm (e.g., [7]) to verify

that the type graph remains acyclic. It is open to question whether Bacon's algorithm coupled with explicit recursion or Alpern's algorithm coupled with dynamic types would provide a more efficient solution to the problem of using containers that recurse to arbitrary depths.

6.0 Conclusions

Adding container types to a language would permit the use of reference counting for automatic garbage collection, which when coupled with an optimizing compiler and region-based memory management has the potential to make automatic memory management possible in hard real-time computing. Container types do not restrict the programmer if they are coupled with the ability to define types at run-time, and when used in a language that only permits static type definitions the addition of container types will only restrict the programmer's use of containers that recurse to arbitrary depths.

7.0 References

- [1] Greg Bollella, Ben Brosgol, Steve Furr, David Hardin, Peter Dibble, James Gosling, and Mark Turnbull. *The Real-Time Specification for Java™*. Addison Wesley, 2000.
- [2] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.
- [3] J. Weizenbaum. Recovery of reentrant list structures in SLIP. *Communications of the ACM*, 12(7):370-372, July 1969.
- [4] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269-273, July 1980.
- [5] David F. Bacon, V.T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. Proc. European Conf on Object-Oriented Programming, June, 2001, LNCS vol. 2072, 207-235.
- [6] David Gay and Alex Aiken. Language Support for Regions. *ACM SIGPLAN Notices*, 36(5):70-80, May 2001.
- [7] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, F. Kenneth Zadeck. Incremental evaluation of computational circuits. *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. 32-42. 1990.