Reflective Garbage Collection

Barry Hayes

1.0 The Way We Were

When garbage collection was first introduced, the world was a simple place -- a collector only had to work in a single language, few algorithms were known and they were all fairly simple, data sets tended to be small, the semantics of collection were exactly that it be semantically invisible, and users didn't expect much performance from garbage collected programs.

2.0 The Way We Are

Many things have happened in the meantime to make my job harder. Garbage collection is now often viewed as a system-wide problem, and collectors must work on networked, multi-threaded programs that contain active agents. The algorithms have gotten so complex that I don't feel I can easily explore the design space, write the algorithms well, or co-opt decisions that the user might be better at making.

Heaps have gotten huge in a spiral with improved algorithms -- users expect more, designers hack better algorithms, and so users expect more. But sophisticated users know that garbage collection is no longer an excuse for poor performance.

The increased demands cause designers to turn to tweaking, writing code in silly low-level languages, and at the same time trying to get more and more semantic information from the system being collected so it can try to use more complex algorithms for better efficiency. Collectors are not complex programs at their base, but constant improvements have made many of them quite brittle.

Also, the semantics of garbage collection has changed. Many modern systems include some kind of finalization, where some of the actions of the collector are visible at the user level in a controlled way. Most finalization semantics work about equally well, and cover almost all the cases users stumble across. *Almost* all. And who gets all their tears and tales of woe when they fall into a situation that cries out for a change to the finalization semantics? Me and my pals. And I, for one, am sick of it.

3.0 The Way We Ought To Be

This seems to be a situation that is ripe for some kind of relief. Garbage collectors should really be written in high-level languages. Between the compilers and the algorithms, designers can now deliver default collectors that meet response criteria Building in a reflective structure would let users modify or extend the collector as needed..

The next sections detail a few places where reflection might be particularly useful.

3.1 Generational Collection

Most collectors these days are generational. They partition the objects by age and attack the young objects more aggressively. The answers to certain key questions -- When to collections happen? When does an object get promoted? -- depend on semantics of the program that the collector does not know in advance --What sort of pause times are acceptable? How cheap or dear is memory? How large will a "typical" data set be for this application?

Some collectors provide a few simple knobs for users to twiddle to control some parameters -- the sizes of old and new space, the tenuring threshold, the full-collection criteria. Sometimes these knobs can only be set at system boot, dooming programs that have changing demands.

In short, most of the collectors that I know of that have some user accessible knobs restrict when they can be twiddled -- the knobs are more of an accident in the design ["Hey! Let's put a knob here."] than *part of* the design.

I'd like to see the collector moved from the system basement to the full light of day. If the basement supplies a way of converting between raw storage and objects, allocaton areas for raw bits, and an area deallocator, most collection styles could be accomidated.

In-place collectors can trace the live objects, setting appropriate bits in either a side-table or a header field. The header fields are nothing special to the system -- they're fields added by the allocater to all objects.

Copying garbage collectors also have a choice over side-tables vs. header fields, but they gain some advantages over most in-place collectors by replicating all live data in a new allocation area and free the old objects en masse by condemning the old area -- thus the need for area-based deallocation.

3.2 The Write Barrier

Generational collectors have an additional requirement; they need to find all the pointers from the old generation into the new generation. Rather than scanning the old generations for pointers, the collector keeps a list of all places in old-space that might point to new-space. On every pointer store, deep in its innards, the system might execute code like this:

```
store(char *ptr, char **loc)
if loc is in old-space
 if ptr is in new-space
 then remember-store
```

Initializing stores into new objects are ignored in the first test, and old-to-old pointers in the second. Neither are needed for generational collections. Deep in its innards, the system knows about the generation structure and the list of cross-generational pointers.

Store checking can be removed by making the objects more reflective about their generation. The `store` methods in young objects can simply do the store. The `store` methods in the old objects can do the second part of the check and remember the store if need be.

Collectors with more complex generation structures also need more complex store checking. When the structure of the generations is the responsibility of a reflective portion of the program, then the store-check is as well. More complex load and store checks also come into play if the collector is running in parallel with other processes -- the store code may depend on what phase of collection is currently active.

Specializing the store-check may actually save cycles in the common case. It also allows collectors with complex generational structures needing complex store-checks to be built at the language level.

3.3 Semantic Input to the Collector

The most interesting improvements in garbage collection these days revolve around using semantic information from the program to improve the garbage collection. Some examples:

- better separate long- and short-lived objects for more efficient collection.
- change trace order for better clustering and paging.
- create a richer and more complex set of generations, e.g. processed-based, or task-based, to improve the collector's ability to focus.
- create new object and pointer classes [like "weak"] to enhance memory semantics.

Rather than burdening low-level code with all of these enhancements, they can be built as reflective code from a few simple primitives, and remain visible, maintainable, and more easily altered.

3.4 More Tracing Applications

The meat of most collectors is an algorithm that takes the transitive closure of the set of root objects to find all reachable objects. Under-the-sheets collectors usually just iterate through the raw bits of an object looking for pointers. It's considered spiffy and cutting edge for the collector to have a type-based map of the object, and avoid checking bits that the type system can certify aren't pointers[1].

These maps are useful for other activities such as making pickles, marshalling RPC arguments, heap migration, and replication. By bringing these shared mechanisms to a reflective level, code sharing will be easier and conflicts will be more apparent.

4.0 Conclusions

Collectors written "above the sheets," reflectively, and in a higher level language, can probably meet the same performance requirements as current low-level collectors. This is due to machine and compiler speed-ups.[2]

Throwing the users power tools will result in a few lost fingers. It would be nice to have a GC toolkit that would permit correct collectors to be built and verified easily.

Without an implementation, it's going to be hard to really tell how well this would work, so I'm going to try one, probably in Self.

1. Modula-3 has these maps at run-time, but doesn't make them available reflectively. They are used by the collector, the pickler, and other "pals" of the system.

2. A quick hack collector for and in Self measured out at about 40kb copied in 100ms on a SPARCStation1. This is fast enough to support a pretty good generational collector.