LittleSmalltalk – The Garbage Collector

Author: Danny Reinhold
Version: 1.0
Status: Published
Date: 19th March 2007

Summary:
Often people ask me how the garbage collection (GC) mechanisms in LittleSmalltalk work and why version 5.x still uses the GC from version 4 and not the simpler looking reference counting mechanisms from version 1.
I try to explain the mechanics and the design criteria that may have lead to this decision (which originally was made by Timothy Budd and not by me) and provide and overview of a possible enhancement.

What is an object in LittleSmalltalk?

Everything in LittleSmalltalk is an object. You may know this, since this holds for all Smalltalk implementations and dialects.
The important characteristics of an object are:

- An object is represented in memory by a flat structure.
- One field in this structure is a pointer to the class of the object (the class also is an object)
- Another field contains a number that tells us how many member variables belong to the object and thus can be used to determine the total memory size of the object
- The remaining fields contain the values of the member variables – since everything in Smalltalk is an object these values in fact are pointers to other objects.

So basically an object is a list of pointers to other objects together with a size information.

Of course there are some special kinds of objects. For example SmallInt objects are in fact stored directly within a value and are no ‚real' objects to speed up the system. But for all objects of user defined classes we can take the definition as given above.

What are the requirements for a good garbage collector?

When we create objects during a LittleSmalltalk session we create lots of memory structures holding mainly pointers to other objects. In other words we are creating a complex object graph.
Most of the objects that are created are only meant for temporary use, like the index variable in a loop. It would be absolutely inacceptable to let those objects stay around in memory forever.We would really soon run into really serious resource problems.

So we need a mechanism that allows us to reuse memory of no longer required objects. This is what the garbage collector (GC) is good for.

A good garbage collector for the LittleSmalltalk system should meet at least these requirements:

- It should be as fast as possible
- It must allow us to release isolated unused objects as well as isolated graphs of objects that are unused but that may have cyclic references within the isolated graph

There are probably much more requirements a good garbage collector should meet but I think that these are the most important.

When LittleSmalltalk was developed the author also had to think about another problem: Allocating and releasing lots of small blocks of memory soon lead to highly fragmented and inefficiently organised memory on many computer systems of that time.
Today malloc(), free() and similar functions are usually implemented in much better ways, but I think it is still better to allocate and release larger amounts of memory seldomly instead of often allocating small pieces…

Simple but imperfect solution

The simplest solution that comes to mind is of course a reference couting mechanism like this:

- Whenever a pointer to an object is stored somewhere a reference counter within that object is incremented.
- Whenever a pointer is overwritten by another value the reference counter of the previously pointed object is decremented
- When the reference counter of an objects decrements to 0 the object gets released from memory

Unfortunately this mechanism fails at both goals: It makes every single operation a little bit slower than it would be without the GC and it is not able to recognize cyclic depending objects – isolated graphs of objects would never be released, although the objects would no longer be required by the system.

How does the garbage collector work in LittleSmalltalk?

The garbage collector in LittleSmalltalk uses three large blocks of memory, the so called *spaces*:

- The static space
- Two dynamic spaces

The static space is a memory area that holds objects that will not be garbage collected at all. From the two dynamic memory spaces there is always one active space and one inactive space.

The *space pointer* is a pointer to the next free byte of memory in the active space.
A new object is always created in the active space. This is simply done by incrementing the space pointer by the size of the object.
Then the class, the size field and the values of the object can be initialized and the object can be used.
There is no special action required during any of the typical operations such as writing a pointer to an object into a member variable of another object etc. Absolutely no special GC related operation! This means that the GC does not affect the usual operation of the LittleSmalltalk system in any way.

But stop! Until now there is no GC involved – just a mechanism for reserving memory for an object.
The object memory must be cleaned up from time to time to avoid exhausting memory usage.

So the garbage collection is realized in an own operation that does not make typical operation of LittleSmalltalk slower in any way, but that needs to be performed from time to time and is quite time consuming then.
In fact the GC is activated either explicitly (when you call a primitive that calls the garbage collection function) or when there is not enough memory left in the active space to allocate memory for the new object.

So what happens when the garbage collection is activated?
The LittleSmalltalk system knows some ‚root' objects that absolutely must be present to have an intact system. And then we state that every object that is referenced directly or indirectly by the root objects is also required to be intact. All objects that are not any longer connected to the graph starting with the root objects are not reachable for the system and may be released.

So we take the root objects and copy them to the inactive space.
The we take all objects that are referenced by the root objects and copy them also to the inactive space. The next step is to adjust the references from the root objects to the other objects to point to the inactive space.
Then lets repeat the process with the object that are referenced by the objects that are referenced by the root objects. You see that this is a simple task of recursion.

When we finished we have copied all objects that are still active to the inactive space and we adjusted all pointers so that they also point to objects in the inactive space.
Usually this means that we didn't copy all objects – the objects that are not connected with the ‚root object graph' have not been copied.
So the inactive space is not as full as the active space and may have enough room for more objects now.

This ist a good time to make the inactive space the active one now and to inactivate the previously active space.

When we see that after this process the new active space still doesn't contain enough free memory to allocate a new object we can resize the inactive space, repeat the garbage collection, resize the active space and make the inactive space the active one.
In effect we have resized both spaces now to hold as much memory as we need and also performed a garbage collection to not take waste with us.

When does the static space get cleaned up?

Above I wrote that the static space is never garbage collected.
This ist rue. But then, you may ask, why don't we run into memory problems all the time and what is the static space good for?

As you saw the garbage collection mechanism as used by LittleSmalltalk doesn't have any influence on the performance of the system as long as garbage collection is not neccessary. But when the system has to collect garbage this process is heavily time consuming.

Now we can ask why should we waste time in cleaning up objects that are never subject to the garbage collector? For example we can be really sure that objects like ‚Class', ‚Boolean', ‚true' etc. are never removed by the garbage collector.
In short we can think that everything that we read at startup from the Smalltalk image will stay in memory for a long time.
So we simply don't load the initial image into the dynamic spaces but into the static space. You remember: The static space is simply a block of memory that never gets garbage collected.
So the garbage collector will only meet objects that really have been created during the runtime of out LittleSmalltalk session.

So when the static memory never gets garbage collected, how can we get objects out of it? Well, we can't! And we don't need to!
When we save the LittleSmalltalk image we do almost the same what we do during a garbage collection. We save all objects starting from some ‚root' objects into the image file. So we save all objects that are reachable directly or indirectly from these root objects and we don't save objects that are isolated or that contain to isolated graphs.

Generational Garbage Collection

A possible extension to the current GC mechanism in LittleSmalltalk is a more general generational garbage collector.
The current mechanism can roughly be thought of as being a two generational garbage collector. The static space is generation 0 and the active space contains all generation 1 objects.
Generation 0 is never garbage collection.

More extended we could specify more generations (that means more separated spaces) of objects, say we have N generations.
Then new objects are created in generation (active space) N-1. When that generation if ‚full' it gets garbage collected. Objects that survive this garbage collection for several times come into generation N-2. When either N-2 is full or when N-1 gets collected m times, generation N-2 gets garbage collected. Objects that survive that process several times then get lifted into generation N-3 etc. And when an object reaches generation 0 it is static and doesn't get garbage collected any more.

This mechanism ensures that the more often occuring garbage collection in generation N-1 will not have to handle as many objects as it is neccessary in the current implementation. So the garbage collection operation would be much faster and it would have a high hit rate, since really many, many of the generation N-1 objects really get ‚released' in each garbage collection call.

So while being a little harder to implement and a little more complex to understand such a generational garbage collection mechanism could improve the performance of the garbage collector.

Summary

Of course this description of the internals of LittleSmalltalk is simplified. But with this knowledge you will hopefully be able to explore and understand the source code.

The current implementation LittleSmalltalk which is mostly the original implementation from Timothy Budd's version 4 doesn't slow down usual operation and works even for highly complex object graphs really well.

Writing a truly generational garbage collector could improve performance even furthor and may be a goal for a future version of LittleSmalltalk. But the current implementation is good enough for all current purposes and will probably be there for quite a long time…