Python Garbage Collector Implementations CPython, PyPy and GaS

Thomas Perl <e0725603@student.tuwien.ac.at>
Seminar on Garbage Collection WS2011/12, TU Wien

January 22, 2012

Abstract

Python is a dynamic language with multiple implementations that utilize different garbage collection mechanisms. This paper compares CPython with PyPy and highlights differences in behavior caused by different GCs. Section 1 discusses Python in general, sections 2 and 3 introduce the garbage collectors of CPython and PyPy. The differences in behavior are discussed in section 4 with examples. An alternative to implementing new GCs in high-level languages (PyPy's approach) is using an external library (GaS, section 5).

1 Introduction to the Python language

Python[1] was created by Guido van Rossum in 1989. Multiple implementations of Python exist today, with CPython (written in the C programming language) being the reference implementation.

In recent years, other implementations were developed by the community: **Jython** (1997) is a Python compiler written in Java and targetting the Java VM, followed by **IronPython** (2006), an implementation of Python for the Common Language Infrastructure (.NET) and most recently **PyPy** (2007), written in RPython (a restricted subset of Python).

PyPy's code can be ran as-is on top of CPython or translated to lower-level runtimes such as POSIX (C runtime) or .NET.

[1] http://www.python.org/

Several modifications to CPython have also been developed in recent years: **Stackless Python** (2000) does not depend on the C call stack, but uses a separate call stack instead. **Psyco** (2003) is a JIT for Python for the x86 architecture. **Unladen Swallow** (2009) adds a JIT using LLVM to Python 2.6.

The CPython implementation utilizes a global interpreter lock (GIL) to synchronize interpreter access – Tabba[3] experimented with adding concurrency to Python using transactional memory support in hardware, and also considered garbage collection-related limitations.

2 Garbage Collection in CPython

Reference counting CPython 1.x used only reference counting for garbage collection. While reference counting is easy to implement and understand, it is not able to detect reference cycles. A cyclic garbage collector has been developed in 1999 and is shipped with Python since version 2.0.

Current status CPython uses reference counting and a generational garbage collector to detect cyles in the current version[2]. Due to reference-counting, a file can be written and closed in one statement without explicitly closing it. This works because the file is flushed and closed as soon as the refcount of the file object (the return value of the *open*() function) becomes zero:

```
open('test.txt', 'w').write('hello world')
```

This, however, won't work in other Python implementations like Jython, IronPython or PyPy, because their garbage collectors will finalize and dispose the file object at some future time (collection time), and only then will the buffers of the file be flushed and the file closed.

To avoid this problem, either explicitly close the file object after writing:

```
# Close the file explicitly with .close()
fp = open('test.txt', 'w')
fp.write('hello world')
fp.close()
```

or implicitly close the file with a context manager[3]:

[2]Python 3.2.2, released September 4, 2011.

[3]http://www.python.org/dev/peps/pep-0343/

```
# Use a context manager (PEP 343)
with open('test.txt', 'w') as fp:
    fp.write('hello world')
```

In both situations, the file object will still only be collected and finalized at some unknown collection time in the future, but it will be closed/flushed at a known point in our code. Both methods also work in CPython.

One might consider the reference-counting nature of CPython useful in some cases, because the developer can be sure that the object is freed as soon as possible. This might also be an advantage in low-memory situations and on embedded or mobile devices.

3 PyPy's Garbage Collection Framework

The PyPy project has developed a "Garbage Collection Framework"[2], which allows Garbage Collection algorithms to be written in a high-level (itself garbage-collected) language and translated into low-level languages like C. This translation has several advantages compared to writing such a virtual machine by hand[1]. PyPy's GCs are written in RPython.

Using high-level languages is just one approach how the development of GCs for existing languages can be simplified – another approach by Wegiel and Krintz[4] will be introduced in section 5.

At translation time, the user can choose between different implementations[4] to be compiled into the resulting Python interpreter, e.g. **Mark and sweep** (classic mark-and-sweep implementation), **Semispace copying** (two-arena garbage collection, copying of alive objects into the other arena happens when the active arena is full), **Generational GC** (implemented as a subclass of the Semispace copying GC, this one adds two-generation garbage collection to distinguish between short-lived and long-living objects), **Hybrid GC** (adding another generation to handle large objects), **Mark & Compact GC** (with in-place compaction to save space, but using multiple passes) and the **Minimark GC** (a combination of the previous methods, rewritten and with a custom allocator).

In addition to the garbage collection framework, PyPy's flexible translation infrastructure allowed the authors to add support for the Boehm GC

[4]the list of garbage collection algorithms available in PyPy can be found at http://codespeak.net/pypy/dist/pypy/doc/garbage_collection.html

and for a reference-counted (but without cycle detection/collection) memory management option[5].

4 Cycles, Finalizers and Uncollectables

Cycle A reference cycle is a cycle of references where a set of objects only has references from inside the set (with each object having a reference count > 0) and no references from the outside. Such objects are not reachable from the outside – they should be freed. Reference counting can't handle cycles:

```
class X:
    def __init__(self):
        self.other = None

a, b = X(), X()              # Refcounts are now 1
a.other = b; b.other = a     # Refcounts are now 2
del a, b                     # Refcounts are now 1
```

After the *del* statement, a and b are not reachable, but their reference count is nonzero, so they can't be freed by reference counting alone.

Finalizer A finalizer in Python (the $\_\_del\_\_(self)$ method) is like a destructor in C++ (but never called explicitly) or $finalize()$ in Java. When the object is about to be freed (when its reference count goes down to zero or the GC collects it), the finalizer is called to clean up.

Uncollectables Reference counting can't detect cycles. For this, another garbage collection mechanism is needed. The generational garbage collector in Python 2 and 3 solves this problem by breaking the cycle at some random point – this only works fine as long as no finalizers need to be called.

Objects in a cycle that has a finalizer will never be collected by the CPython garbage collector, because breaking the cycle modifies object state. Uncollectable objects will appear in the *gc* Python module as *gc.garbage*:

```
import gc

class X:
```

[5] http://codespeak.net/pypy/dist/pypy/doc/translation-aspects.html

```
    def __init__(self):
        self.other = None

    def __del__(self):
        print 'DEL!', self

a, b = X(), X()
a.other = b; b.other = a
del a, b
gc.collect(); gc.collect()
print gc.garbage
```

This will print the following in CPython 2.7.2[6] on an amd64 machine:

```
[<__main__.X instance at 0x7ff5474933b0>,
 <__main__.X instance at 0x7ff5474933f8>]
```

Running the same example on PyPy 1.7[7] yields a different result:

```
DEL! <__main__.X instance at 0x00007fb2c4f09260>
DEL! <__main__.X instance at 0x00007fb2c4f09280>
[]
```

We observe that Python implementations with distinct GCs behave differently: CPython does not even try to get the order of finalizers right, and simply puts uncollectable objects into the global list of garbage for the developer to deal with manually.

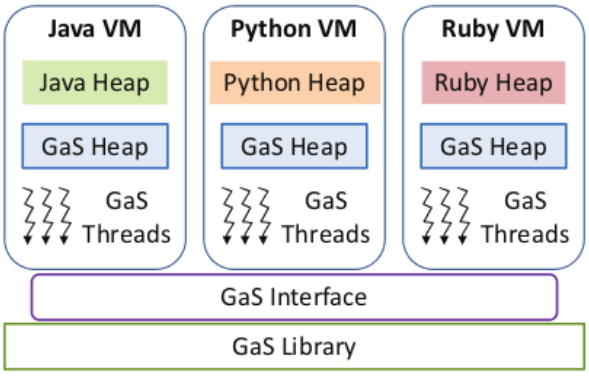
PyPy (with the default GC mechanism "Minimark") provides a "at most once" call guarantee for the finalizer (CPython does not provide such a guarantee at all, not even "at least once"), i.e. the finalizer of any object is called at most once, even if the object is later reincarnated (e.g. by leaking a global reference to *self* in the finalizer). This works because the other object still has a reference to the first object when the finalizer of the first object is called (one could view this as "automatic reincarnation").

Because a finalizer is called at most once, however, the cycle can be removed by simply calling each finalizer once and (after all finalizers in the cycle have been called) freeing the memory of the objects in the cycle (without calling any finalizer).

[6]Python 2.7.2-9 from Debian Testing

[7]Binary from https://bitbucket.org/pypy/pypy/downloads/pypy-1.7-linux64.tar.bz2

Figure 1: Architecture of the GC-as-a-Service library (Source: [4])



The reason why the problem of ordering finalizers is easier to solve in PyPy and not solved in CPython is because the ordering of is more strict and difficult in a mixed reference counting / cycle detection environment than it is in the non-reference-counted garbage collector of PyPy.

5 Garbage Collection as a Service

As we have seen, the high-level description of GCs in PyPy can provide advantages for both developers and users. However, it's not the only approach to save time implementing experimental garbage collectors.

In [4], Wegiel and Krintz discuss their GC-as-a-Service (GaS) garbage collection library (figure 1). The idea is to implement a state-of-the-art concurrent and cooperative garbage collector in a library, with a pre-defined interface that runtimes such as the Java VM, Python VM or Ruby VM can hook into. The garbage collector is non-moving, which means that it supports VMs which assume that objects cannot be moved in memory.

While [1] suggests that one way to implement VMs in an efficient way is to translate high-level languages into lower-level ones, and describe components such as garbage collectors in high-level languages, GaS[4] goes a different route and tries to move the GC logic into a modular library. This also facilitates code reuse, but currently only allows one garbage collector to be utilized, and GaS is not written in a high-level language, which makes it harder to check and implement compared to translation methods as in [2].

GaS is implemented as four-phase GC, where all phases run concurrently:

flag clearing, root dump, object marking and object sweeping. Because the Python integration of GaS uses CPython, GaS has to take reference counting into account, and does so by making the *incref* and *decref* operations conditional, and avoids reference counts to objects allocated on the GaS heap.

GaS's Python integration was done on top of CPython 3.1, and only the implementation of the binary search tree was evaluated. The findings of the experiments were that GaS requires larger heap sizes and also adds some runtime overhead compared to the original GC implementation.

6 Conclusions

The research project **PyPy** includes a framework for experimenting with different garbage collection algorithms that are more sophisticated than the GC implemented in **CPython**. An alternative to the high-level GC framework of PyPy is **GaS**, a pluggable GC library.

A short overview of the different garbage collection algorithms in PyPy was given, followed by an example analysis of different behaviors. CPython and PyPy have different semantics and guarantees when and how often the finalizer is called on an object.

Conclusion 1 For historic and practical reasons, implementations of Python have different behaviors and semantics when it comes to garbage collection and finalizers. Developers writing portable code have to be aware of these, and must not rely on implicit behavior of any single implementation.

Conclusion 2 To simplify the development of new garbage collectors for existing languages, garbage collectors can be written in high-level languages and translated to a target runtime (PyPy's approach) or written as external library providing specified interfaces to VMs (GaS' approach).

References

[1] C. F. Bolz and A. Rigo. How to not write virtual machines for dynamic languages. In *Proceeding of Dyla*, 2007.

[2] C. F. Bolz and A. Rigo. IST FP6-004779: researching a highly flexible and modular language platform and implementing it by leveraging the open source python language and community, 2007.

[3] F. Tabba. Adding concurrency in python using a commercial processor's hardware transactional memory support. *SIGARCH Comput. Archit. News*, 38:12–19, April 2010.

[4] M. Wegiel and C. Krintz. Concurrent collection as an operating system service for cross-runtime cross-language memory management, 2010.