Ephemerons meet OCaml GC

François Bobot
CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France

1 Introduction

Garbage collectors (GCs) manage the memory for the programmers and help to ensure the safety of the programs by freeing memory only when it cannot be used anymore. GCs detect that a memory block can't be used when it is not reachable through pointers from memory blocks that are used. In order to allows the GC to free memory blocks and to avoid memory leaks, programmers must take care to not keep reachable any memory block which is not useful anymore.

A difficulty arises when one wants to cache or memoize the computation of a function because one wants to keep the result as long as the argument and the function are used. In this presentation we call this the requirement. The first part of the presentation will show that it is not possible to find a general satisfactory solution for this problem with the current OCaml GC by going through tentative solutions with increasing complexity. Then we will present how the OCaml GC can be extended with ephemerons to solve the problem.

2 Tentative Solutions with usual GC

Memoization is a programming technique that keeps for a function f the result V of the first application with an argument K in order to return V without computation the next times the function is called with K. In the first part of the presentation we are interested in memoizing a function without creating memory leaks when the argument or the function can be freed.

Naive Solution An obvious solution consists in using a traditional dictionary data structure T (hash table, balanced tree, etc.) mapping keys K to values V. This solution (Figure 1a) has the following major drawback: as long as T is reachable, all keys and values bound in T are also reachable. Therefore the requirement does not hold. Conclusion: T should not hold pointers to keys, the arguments of f.

Indirect Keys In order to avoid keeping pointers to keys in the table, we can use indirection. We points from the T to the key K using a weak pointer. A weak pointer do not count for the reachability. For the binding to be removed from the dictionary when K is reclaimed, we attach a finalizer that removes the binding from T. A finalizer is called when the value to which it is attached is going to be reclaimed. In order not to create an additional path to T, the finalizer only holds a weak pointer to T. Thus, if T is reclaimed before K, the finalizer will do nothing. At first glance, it seems to be a satisfactory solution. But let us suppose that K is reachable from V (e.g. $V = K$) (Figure 1b). Then we obtain exactly the same situation as with the previous solution: K is reachable from T and the requirement does not hold anymore. Conclusion: T should not hold pointers to values either.

Role Inversion Apparently we cannot stock bindings inside tables. What if we keep them in keys instead? Inside each key, we put a dictionary mapping "tables" (from now on, just a unique identifier) to values. Basically, we just swap K and T in the previous solution (Figure 1c). One obvious limitation of this approach is that the programmer must have control over the type of keys, which become now mutable data structures. However, there is no observable difference w.r.t. the usual situation where the table is a mutable dictionary. The key K being reachable from V is not a problem anymore, since the whole cycle involving K and V can be reclaimed. Of course, if T happens to be reachable

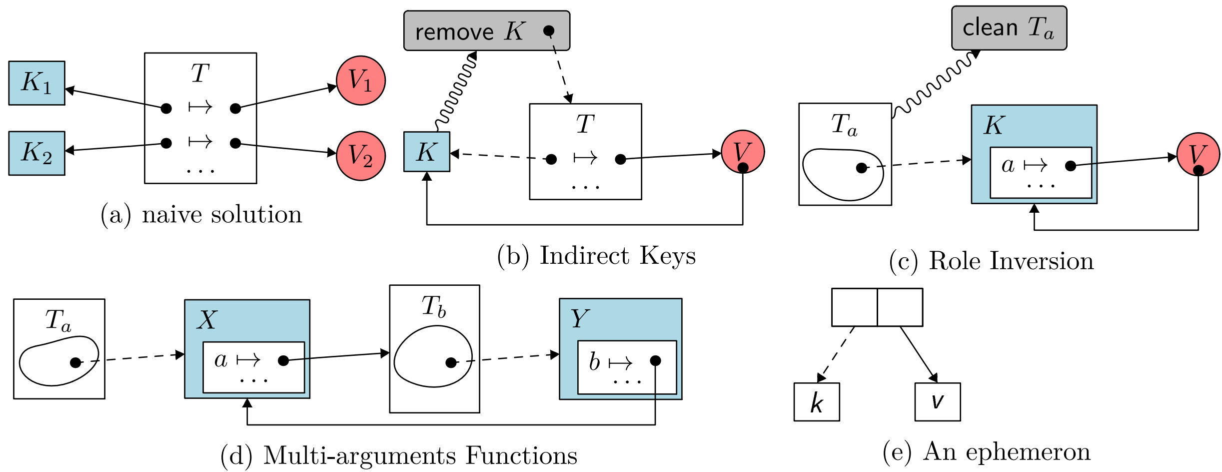

Figure 1: Schema of tentative solutions and ephemerons

from V, then the requirement does not hold, for the same reasons as before. This annoying situation may appear in practice when a multi-arguments function such as $\mathbf{K}(X, Y) = X$ by is memoized using partial applications (Figure 1d).

Indeed I proved that it is impossible to design a data structure with usual pointers, weak pointers and finalization that will behave as expected in every situation, but I will not present it during the presentation due to lack of time.

3 Ephemerons

Hayes describes similar problems and a new kind of value that he named ephemeron [1]. In its simplest form (Figure 1e) an ephemeron is a memory block that contains two pointers. The first one weakly points to the key K. The second points to the value V in a special way: V is said to be reachable iff the ephemeron and the value K are already reachable. That is the key point: an ephemeron allows the programmer to express a conjunction whereas usual GCs allows only one-to-one relations. So it is a strict improvement in expressivity.

With this basic block one can define perfect weak hash table where the data is kept until the key or the hash table is not reachable anymore even in presence of cycles. That allows us to implement a memoization that verifies the initial requirement.

The presentation will describe how the OCaml GC has been extended for providing ephemerons and how the worst costly cases have been mitigated. Interestingly, OCaml weak pointers can be implemented with generalized ephemerons (arbitrary number of keys but one value). At the time of the presentation this work[1] should be merged in the OCaml's trunk. The choices made for the interface of the module `Ephemeron` will be discussed and benchmarks using a consequent OCaml programs (the why3 platform) will be provided.

Finally the presentation will compare this implementation with the weak pointers [2] of Haskell which results of different design choices.

References

[1] Barry Hayes. Ephemerons: A new finalization mechanism. In Mary E. S. Loomis, Toby Bloom, and A. Michael Berman, editors, *OOPSLA*, pages 176–183. ACM, 1997.

[2] Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in Haskell. In Pieter W. M. Koopman and Chris Clack, editors, *IFL*, volume 1868 of *Lecture Notes in Computer Science*, pages 37–58. Springer, 1999.

[1] `https://github.com/ocaml/ocaml/pull/22`