Hashconsing in an Incrementally Garbage-Collected System

A Story of Weak Pointers and Hashconsing in OCaml 3.10.2

Pascal Cuoq [*]
CEA LIST, Software Reliability Labs,
91191 Gif-sur-Yvette Cedex, France
Pascal.Cuoq@cea.fr

Damien Doligez
INRIA
Domaine de Voluceau, BP 105
78153 Le Chesnay, France
Damien.Doligez@inria.fr

Abstract

This article describes the implementations of weak pointers, weak hashtables and hashconsing in version 3.10.2 of the Objective Caml system, with focus on several performance pitfalls and their solutions.

Categories and Subject Descriptors D2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Design, Performance

Keywords OCaml, garbage collection, weak pointers, weak references, weak hashtables, hash-consing, maximal sharing

1. Introduction

This article describes the implementations of weak pointers, weak hashtables and hashconsing in OCaml. None of these concepts are novel. Weak pointers have existed in some LISP implementations for a long time (Haible 2005; Goto 1974). The subtitle refers more precisely to OCaml's implementation of these features, which is much more recent (Leroy 1997). Despite the long history of weak references, subtle performance issues were noticed in the first OCaml implementation when it was confronted with real use. The 3.10.2 version of the OCaml system (Leroy et al. 2007) fixes all the issues that have been encountered at the time of this writing. Hashconsing in OCaml (Filliâtre and Conchon 2006), based on the provided weak arrays and weak hashtables, now works reliably and is employed in heavy duty applications, such as automated theorem proving (Bonichon et al. 2007; Conchon and Contejean 2006) and abstract interpretation analysis (Frama-C development team 2008). What is more, these changes have gone unnoticed by most OCaml developers, even the developers of some of the aforementioned heavy duty applications, because there has not been any programmer-visible interface change associated to the fixes that were taking place under the hood. In fact, weak pointers in a functional language, when they are provided expressly to allow hashconsing, are difficult to get right for a conjunction of two reasons. Firstly, a wide class of bugs in the implementation of weak pointers only cause some values to remain in memory for too long, and have no other, more obvious, ill side-effects. Secondly, it is hard to predict the expected speedup when adding hashconsing to a program, which makes it difficult to tell that it is less than it could have been. This article describes the issues that were found and the solutions that were provided in the process of making OCaml's weak hashtables, and the underlying weak arrays, scale up.

This article provides some insights into OCaml's Garbage Collector's implementation. This may be of interest to any OCaml programmer who ever interfaced OCaml and C code and was left wondering why the Ocaml reference manual needed to be so strict on the subject. However, the issues described here are not specific to functional languages, they should be applicable to other languages that provide weak references such as Python (Beazley and Rossum 1999) or Java (Gosling et al. 2000).

The considerations to take into account when making the choice of a Garbage Collection technique are described in (Wilson 1992). Following the terminology used there, the Garbage Collector (GC) implemented in the current versions of OCaml is both generational and incremental.

This article starts with an overview of the inner workings of OCaml's GC (section 2), providing context and vocabulary for the rest of the discussion. The "weak pointer" feature is described in section 3, while the implementation of this feature in OCaml's GC is discussed in section 4. OCaml provides a higher-level construct built upon weak pointers, weak hashtables. The interface of these weak hashtables is described in section 5. An example of use of these weak hashtables for hashconsing is provided in section 6, and in section 7, this example is used to illustrate the pitfalls that should be avoided when implementing weak hashtables. Section 8 describes a different possible implementation for weak hashtables, and section 9 benchmarks the two implementations, each with both OCaml's 3.09.3 and OCaml's 3.10.2 runtimes.

2. Garbage collection in OCaml

The heap of an OCaml program is divided into a *minor heap*, whence newly requested blocks are always allocated provided that they are small enough, and a *major heap*, where live blocks from the minor heap are copied when the minor heap is full.

The blocks that are allocated in both heaps contain the number of words that was requested by the ML program that allocated them — for instance, two words for a *Cons* cell — plus a header word that the GC uses for bookkeeping. The header word is divided in bitfields respectively dedicated to storing the *size* of the block, a *tag* containing some rough type information about the contents of the block, and a 2-bit *color* used by the GC for tracking reachable

[*] This work has been supported by the french RNTL project CAT ANR05RNTL00301

ML'08, September 21, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-062-3/08/09...$5.00

blocks. The files implementing OCaml's GC can be found in the directory `byterun` in OCaml's distribution (Leroy et al. 2007).

For scoping reasons, because the blocks in the minor heap are more recent than the blocks in the major heap, one might think that there would be no references from the major heap to the minor heap. There can actually be a few such references, firstly because it is not exactly true that blocks in the minor heap are always more recent than blocks in the major heap: big blocks are allocated directly into the major heap, and these can reference all the blocks already allocated in the minor heap at that time. Another reason why there can be references from the major heap to the minor heap is that OCaml allows some values to be mutated: a cell in the major heap may be modified in-place and the address of a more recent block in the minor heap can be written there. In the implementation, these two causes for the existence of pointers from the major heap to the minor heap are the same one, because big blocks are initialized using the same `Modify` macro that also serves for in-place modifications.

When the minor heap is full — when there is no free space left to allocate new blocks from — a *minor collection* takes place. A minor collection makes space available again in the minor heap by copying the live blocks from the minor heap to the major heap. Since it follows that everything left in the minor heap is dead, the GC can then re-use the minor heap in its entirety.

The addresses of locations in the major heap where a pointer to a block in the minor heap has been written are called the *remembered set* (Wilson 1992). In OCaml's implementation, the remembered set is stored in a structure called `ref_table`. Keeping this table up-to-date is one job of the `Modify` macro. Maintaining this table of locations is necessary to determine which blocks in the minor heap are alive at the time of doing a minor collection, and also in order to be able to update these locations when the blocks are consequently moved to the major heap. At the time of doing a minor collection, the live blocks are found by recursive traversal, starting with the `ref_table`, and the part of the stack that is recent enough to possibly contain pointers to the minor heap. The major heap does not need to be examined in its entirety in order to do a minor collection.

The major heap is garbage-collected with what is essentially an incremental *Mark and Sweep* algorithm. Slices of *major collection* are done just after each minor collection.

Figure 1 shows a simplified view of the memory of an OCaml program. If a minor GC is triggered in this state, blocks A and C will be copied to the major heap (and the root references to them will be updated); block F will also be copied to the major heap, and the reference inside L will be updated; then the whole minor heap will be cleared (leaving B, D, E, and G behind), as well as the `ref_table`. Note that J is not reclaimed at this point, since it is in the major heap and thus not subject to minor collection. Only the major GC can deallocate it.

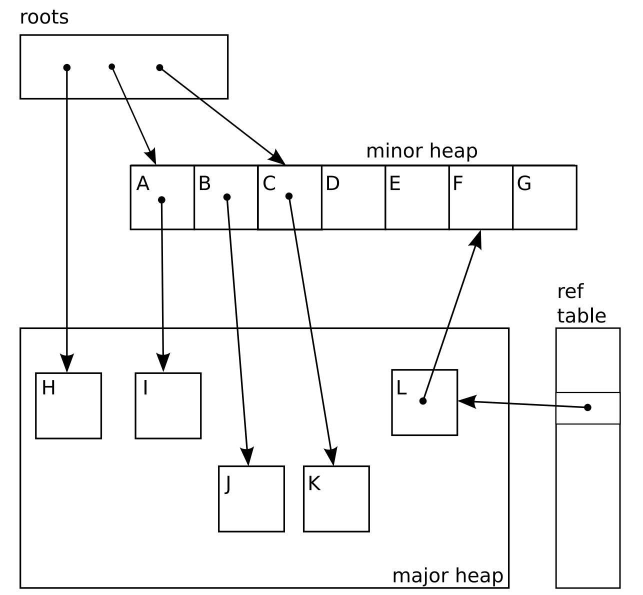

Figure 1. OCaml memory layout

3. Weak pointers, weak arrays

Weak pointers (Haible 2005) constitute one of the advanced features that a GC can have. They provide the possibility of recording a pointer to a block without making the block automatically alive. Of course, when accessing such a pointer, the program may be notified that this block has been reclaimed, which may cause some interesting race conditions. In OCaml, the function `Weak.get` that accesses a weak pointer to an object of type `t` returns a value of type `t option` — that is, one of `Some(v)` or `None`. When `Weak.get` returns `None`, it means that the GC has determined that the value referenced by the weak pointer had become unreachable by conventional pointers, and has reclaimed it.

An often mentioned example of use of weak pointers is that of the implementation of a "smart cache" (Chailloux et al. 2000), that allows the GC to reclaim the memory used by the cache "when necessary". Although this example is the first one that springs to mind when reading the (somewhat dry) corresponding section of the reference manual, it should be remembered as an example of how not to use weak pointers. Indeed, the GC is always reclaiming memory! It does not know, and it is not its role to care, if the cache is actually very small compared to the amount of memory available from the system, and if it makes the program much faster. It will bluntly reclaim the memory used by the so-called smart cache when the time comes. Additionally, there are no guarantees at all concerning the time it takes for a dead block to be reclaimed by most GCs. While having an upper bound on this time is desirable and some specialized GCs attempt to provide one (Baker 1978), it does not make much sense to expect a lower bound on this time. Improving the GC so that it would reclaim any dead block very quickly without unreasonable overhead would be an improvement made in good faith — and probably with a good bit of thinking. Still, such an improvement would slow down any program that relies on weak pointers for caching.

Useful examples of the use of weak pointers are examples where the values referenced by weak pointers may also be referenced through non-weak pointers. Let us here emphasize that blocks that are referenced by weak pointers are not reclaimed if they are also referenced normally. This gives us our first valid example of use of weak pointers, the poor man's *finalized block*. *Finalization* (Leroy et al. 2007) is another advanced feature that a GC can provide. It gives the possibility to execute a specific function when a block is reclaimed. If a GC does not provide this feature but provides weak pointers, it is possible to emulate finalization in the case where the finalization function does not need to get the (dead) block as argument, by keeping a weak pointer to the block to be finalized. Its disappearance can then be detected by polling. Trying to do this with a normal pointer would result in keeping the block alive, which would defeat the purpose of finalization.

Another, more direct example of the usefulness of weak pointers can be found in the implementation in any garbage-collected language of an interpreter for a garbage-collected language (Peyton-Jones et al. 1999; Elliott and Hudak 1997). In this case, it is possible for the interpreted language to piggy-back onto the host language's memory management, saving the interpreter's implementer the trouble of writing his own GC. The fact that the interpreted language's values (say, synchronous processes) are simply values from the host language makes this implementation scheme even more desirable. This gives the two languages a chance to collaborate closely. For instance, the standard library of the host language is thus conveniently available from the interpreted language.

However, if the interpreted language needs to maintain a list of living processes, a difficulty appears with the desirable implementation scheme proposed above. Keeping this list can be necessary for any kind of bookkeeping — for instance, to inform all living reactive processes that a synchronous clock shared by all of them has advanced. The intention, however, is likely to be that "only living (i.e. referenced) processes should receive the signal", as opposed to the strategy of "keeping every process alive so that it can receive the signal". The latter is a kind of memory leak. It is exactly what happens if an ordinary, non-weak data structure is used to store the list of processes. Weak pointers are the standard solution to this difficulty: in order to allow processes to be garbage-collected when they are no longer referenced from other processes, the data structure where they are enumerated should use weak references to them.

Hashconsing is another example of the use of weak pointers. Hashconsing is a technique that employs a hashtable to remember all the value of a given type `t` that have been created. This way, it is possible, when the program is about to create a new value of type `t`, to check in the hashtable if this exact value does not already exist, and to use the existing value if it does, thus ensuring *maximal sharing*. In some applications, hashconsing can improve both memory usage and computation times because the maximal sharing property also makes it cheap to determine if two values are equal. However, when hashconsing is implemented without taking care of using weak pointers, there is a memory leak problem: the hashtable then causes all the values of type `t` to remain in memory even when they are no references to them except the one from the hashtable. But in fact, the maximal sharing property does not require to keep every value of type `t` in memory indefinitely! When one such value has ceased to be useful, it can safely be collected, because it can always be created again later. For this reason, weak references should be used at some point when implementing hashconsing. Of all the uses for weak pointers, hashconsing will receive the most attention in this article.

In OCaml, there is a two-word memory overhead for allocating a single weak pointer, but this overhead does not increase for additional weak pointers in the same block. The primitive construct provided is therefore that of *weak arrays*, so that, when it is not otherwise too constraining, several weak pointers can be allocated at once in a single array. The functions for manipulating weak arrays can be found at toplevel within the `Weak` module in OCaml's standard library.

4. Adding weak arrays to OCaml

This section describes the issues raised by the addition of weak arrays to the OCaml system.

4.1 Interface of the `Weak` module

In OCaml, `'a Weak.t` is the type of weak arrays of `'a`. Such arrays can be created, modified and accessed with the following primitives:

```
type 'a t
val create : int -> 'a t
val set : 'a t -> int -> 'a option -> unit
val get : 'a t -> int -> 'a option
val get_copy : 'a t -> int -> 'a option
val check : 'a t -> int -> bool
```

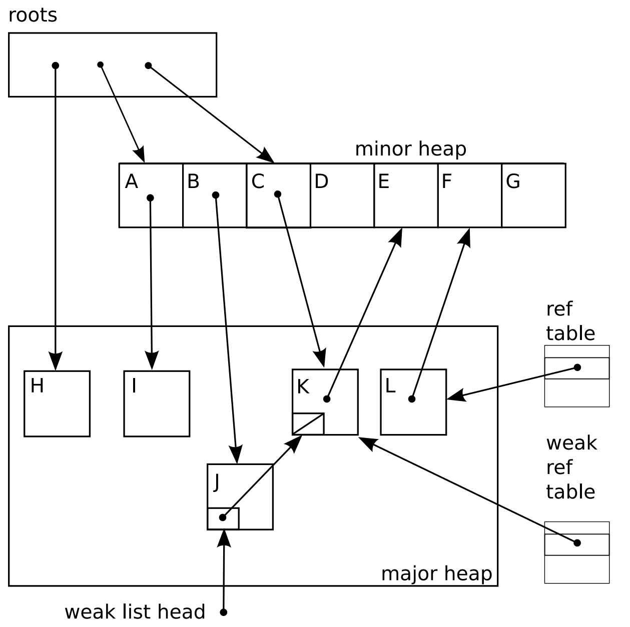

Figure 2. OCaml memory with weak arrays

The function `get_copy` returns a shallow copy of the stored value. The utility of this function will be explained in section 7.2.

4.2 Implementation

Weak arrays are allocated directly in the major heap for simplicity. They are materialized as blocks with the `Abstract_tag`, which means that the contents of these blocks are not traversed recursively by the major collection. The `Abstract_tag` has always existed in the current implementation of OCaml's GC. It is habitually used for data (such as floating-point values) whose bits should not be treated as though they composed an address. In the case of a weak array, the values inside the block are indeed addresses, but they should not be traversed by the GC because that would make them alive, which is precisely what is not wanted. If there are other, normal pointers pointing to a block referenced from a weak array, it is the traversal of these pointers that will make the block alive. If there aren't, the block will eventually be reclaimed as dead and the value of the weak pointer changed to a sentinel that means the block has become dead.

The first word of each block representing a weak array is reserved for chain-linking all the weak array blocks together. This is illustrated in figure 2, where J and K are the weak arrays. Note that because of the `Abstract_tag`, this chain-linking does not make the weak array blocks automatically alive. The liveness status of a weak array block is determined like that of any other block, by the presence of a live conventional pointer to it in the heap. The contents of the block representing a weak array, from the second word until the last one, are the actual weak references. A sentinel, `weak_none`, is used to indicate that a particular weak reference was referencing a value that has been reclaimed.

The fact that the weak array blocks are chain-linked together allows to traverse them in a special phase of the garbage collection that takes place right after the major collection's *Mark* phase is finished. At that point during the major GC cycle, the blocks that are going to be considered live and those that are going to be reclaimed are clearly marked as such in the *color* field of their respective headers. It is during this special phase that the weak pointers found to be pointing to *white* blocks — blocks that are going to be reclaimed as dead in the upcoming *Sweep* phase — are replaced by the sentinel `weak_none`.

Besides, live blocks may also be moved around by the major GC in a phase called *Compaction*, and unlike `ref_table` in the case of the blocks moving from the minor heap to the major heap, there isn't a convenient complete list of locations of pointers to update. The GC knows how to correctly update all the pointers to moving blocks that it has traversed, but since it does not traverse the weak array blocks at all while examining the heap, this needs to be done in another specific phase by traversal of the weak array chain.

4.3 Pitfalls

The description above corresponds exactly to Ocaml 3.09.3's implementation: when a pointer into the minor heap is written at any location in the major heap, the location is added to the `ref_table` so that it can later be updated when the pointed block is moved. Since weak arrays are mutable structures allocated in the major heap, the above conditions apply in particular each time that a value from the minor heap is added to a weak array. In Ocaml 3.09.3, the location inside the weak array would be added to the `ref_table` like any other major heap location updated with a pointer towards the minor heap. However, this could incur a significant memory waste in some circumstances. This section explains why.

The major GC is incremental. When blocks are copied to it from the minor heap, these blocks are automatically marked as alive. In normal circumstances, they are alive, for the major GC's definition of liveness as "reachability at the beginning of the major cycle" — since they have been copied, they definitely were reachable. Except, of course, for blocks in the minor heap whose only claim to reachability was a reference in a weak array. For these, the usual algorithm of "copy block to the major heap, mark block as alive and update the pointing location", while not being observationally wrong, is sub-optimal. These blocks, if not referred to through a normal pointer (that would either be registered in the `ref_table` or that would have been discovered by the minor heap traversal) can rightfully be considered as dead, because there are no non-weak references to them at that instant. For these blocks, the best algorithm to apply is "reclaim the block and change the weak pointer accordingly". Applying the usual algorithm to them means that they will stay in memory on average one major cycle and a half longer than necessary.

In OCaml 3.10.2, in order to handle the case of these weak references more appropriately, the locations inside weak arrays being updated with a pointer to the minor heap are recorded in a different structure, the `weak_ref_table`. Only the locations in the `ref_table` are treated as roots for the minor collection. Right after the minor collection is finished, the `weak_ref_table` is analyzed in an additional step. Locations recorded in the `weak_ref_table` that contain the address of a block that was moved to the major heap are updated with the new address, while the locations that contain the address of a block that was not visited during the minor heap traversal are marked as being weak pointers that have been reclaimed.

Let us go back to figure 2 for an example. In this figure, K is a weak array and the `weak_ref_table` points to it. K points to E, but the minor collection will not copy E into the major heap, but leave it to be reclaimed along with B, D, and G. Then the additional phase will erase the pointer from K to E, replacing it with `weak_none`.

Note that the implementation of the `weak_ref_table` shares most of its code with the existing `ref_table`, hence this new feature does not add significant complexity to the runtime system.

5. Adding weak hashtables to OCaml

The weak hashtables described here provide slightly different functionality than typically found in weak hashtables in other languages. Indeed, OCaml's weak hashtables are implemented without relying on any primitive feature other than weak pointers, whereas many variations on the theme of weak hashtables have been defined (Haible 2005; Li 2007), some of which require more support in the language core than simple weak pointers. OCaml's weak hashtables are not hashed maps but hashed sets, where only the keys are stored. While this means that some sophisticated usages of weak pointers are disallowed, it will be seen in section 6 that the drawbacks when implementing hashconsing are minimal.

This is the interface of weak hashtables:

```
module type S = sig
  type data
  type t
  val create : int -> t
  val merge : t -> data -> data
  ...
end
module Make (H : Hashtbl.HashedType) :
                                S with type data = H.t
```

The functor `Weak.Make` takes a module `H` that contains a type `t` of the data to be hashconsed, and functions `hash` and `equal`; it creates a weak hashtable of `H.t`. It provides, among others, a `create` function to allocate new hashtables, and a `merge` function to look up or add a piece of data in a given hashtable. This functor is part of the OCaml standard library, and it is written entirely in ML, relying only on the "weak arrays" primitive. A weak hashtable, like a normal hashtable, is represented as an array of buckets. The basic idea is to represent each bucket as a weak array containing the values that have been `merge`d into the table at this point. The function `merge` applies the hash function to the passed value `v`, and infers from the result which bucket `v` should go into — or already is in. It can then compare `v` to the values already stored in the bucket in order to decide which case applies, return the found value `v1` if it finds one such that `(equal v v1)` is true, or add `v` to the bucket and return the value `v` otherwise.

OCaml's weak hashtable implementation distinguishes itself from some others (Peyton-Jones et al. 1999; Haible 2005) in that it allows arbitrary ML functions to be used as the equality and the equality-compatible hash function on keys. In practice, the OCaml weak hashtable library is presented as a functor to be applied to a module that provides these functions. For instance, assuming that the module `L` is a module of lists with the `Hashtbl.HashedType` interface (i.e. providing a type `t` and functions `hash` and `equal`), a weak hashtable of lists can be created with:

```
module ListWeakHashtbl = Weak.Make(L)
```

This feature is vital, for example in the Zenon theorem prover (Bonichon et al. 2007), to make weak hashtables of first-order terms modulo alpha-conversion, and in Frama-C (Frama-C development team 2008) where AVL trees are stored in weak hashtables, and must be compared modulo rebalancing.

Note that it is not necessary for a programming language to have a sophisticated module system in order to provide the feature "weak hashtable with user-defined equality and (equality-compatible) hash function". Any language where functions are first-class citizens could provide a similar feature. Still, some of the pitfalls to be described in section 7 come from the fact that the equality and hash functions are ordinary, user-defined functions. The authors did not notice these pitfalls mentioned in the literature, perhaps because this feature was never provided together with incremental garbage collection.

6. Hashconsing

In this section, we will illustrate the use of OCaml's flavor of weak hashtables with the example of the hashconsing of a list type. The explanations given here are solely for the purpose of showing one concrete use of weak hashtables. The implementation of hashconsing in OCaml is described in much more detail in (Filliâtre and Conchon 2006).

Creating a hashconsed list type is little more than defining an adequate `cons` function that takes a head `h` and a tail `t` and creates the list `Cons(h, t)`. The `cons` function should guarantee that only one instance of the `Cons` constructor applied to a given (`h`, `t`) pair is visible at any time. In other programming languages, this effect can be obtained by use of a weak mapping from `h` and `t` to the constructed list. In OCaml, because the weak hashtables are in fact hashsets, the `Cons` constructor should be applied tentatively to form a list `l` similar to the one that is desired, and then the weak hashtable should be looked up for a value that is "`equal`" to `l`.

```
let cons h t =
  let l = Cons(h, t) in
  ListWeakHashtbl.merge tbl l
```

In the rest of this section we will slightly abuse the definition of OCaml's physical equality primitive ==, and assume that it is possible to use it to compare two values if both these values have been hashconsed in the same table. This abuse is widespread, but it still is an abuse: for immutable objects, the only formal guarantee concerning physical equality is that it implies structural equality. In theory, the OCaml implementation reserves the right to share or duplicate structurally equal immutable values. In practice, we know that == has the behavior we need on the `Cons` cells. A better implementation would not use == for comparing lists, but instead either arrange to physically compare mutable objects (for which == provides more guarantees) or tag the applications of `Cons` with unique integers. In our example, the abuse allows for conciseness in the definition of equality on hashconsed lists as the physical equality:

```
let list_equal l1 l2 = (l1 == l2)
```

The equality to use during weak hashtable lookups should not be the equality defined above. Indeed, during a lookup, one of the values in the comparison is a `Cons` that was applied without knowing yet whether this application was redundant. On the other hand, the equality to use for the lookup in the weak hashtable can assume by induction that the hashconsed children of its argument nodes are equal iff they are physically equal. In other words, this equality, which is only useful for lookups, can be defined as in the following module:

```
module L =
struct
  type t = Empty | Cons of value * t
  let equal l1 l2 =
    match l1, l2 with
      (Cons(h1, t1)), (Cons(h2, t2)) ->
        (list_equal t1 t2) && (element_equal h1 h2)
    | ...
  let hash l = ...
end
```

This module `L` is the one that should be passed to the functor `Weak.Make` in section 5. The `hash` function inside the module `L` should be defined with identical care for identical reasons. If the programmer's decision was to tag the applications of `Cons` with unique integers, these tags make a fine hash function for hashconsed lists, but the function `L.hash` should not use its argument node's tag as a hash value, because its argument has not been hashconsed yet (the `Cons` constructor was applied tentatively with a fresh tag). Instead, the function `L.hash` should be defined so as to make use of the tags of the children of the node it is applied to. Similarly to the function `L.equal`, it does not need to look deeper into its argument than the first level.

To reiterate, while the module `L` defined here should be the one passed to the functor `Weak.Make` in order to create a weak hashtable module able to soundly store and recover lists being hashconsed, its functions should not be used for anything else, as there are better implementations available when the argument(s) are lists that have already been hashconsed.

7. Pitfalls in the implementation of weak hashtables

Along with the performance issues already mentioned in the treatment of weak arrays by the runtime system, the introduction of hashconsing in Frama-C revealed issues that were specific to the weak hashtable layer. A common theme underlying these issues is that the resemblance of weak hashtables to standard hashtables at the interface level is misleading: performance can be suboptimal if weak hashtables are implemented too similarly to standard hashtables.

7.1 Deciding when to resize weak hashtables or buckets

OCaml's standard hashtables, as well as weak hashtables, are able to resize themselves dynamically as the number of elements they contain is growing. For a standard hashtable, the decision to resize is easy to take: it is only a matter of counting the elements that go in and come out, and to allocate a new, bigger array of buckets when a fixed average number of elements per bucket has been reached. In the case of a weak hashtable, however, the problem is more complicated. What the standard hashtable is doing, by counting the elements that go in and come out of the table, is to keep a count of the number of elements inside the table. In the case of a weak hashtable as used for hashconsing, the elements come in and eventually evaporate silently. At cruising speed, the quantity of live data inside the hashtable remains about constant while new values come in all the time, so that for a program running for an unbounded time, the total amount of data that has been put inside the table may be unbounded. But without additional support from the GC, it is not known how many live elements are inside the weak hashtable and therefore it is difficult to make an informed choice about resizing.

The same problem exists at the level of the individual bucket. A strategy that may seem both reasonable and simple to implement is the following: grow each bucket when it needs to, otherwise keep them the same size. This strategy does not work so well for the hypothetical program above that runs an unbounded time while always keeping a bounded amount of live data inside the table. Because of inevitable statistical fluctuations, each bucket will at one time or another grow past any fixed size — and then empty itself back as the pointers are reclaimed and the statistical tide moves to another bucket. Eventually, the weak hashtable implementing the "reasonable" strategy will have arbitrarily large buckets containing almost exclusively reclaimed weak pointers.

In OCaml 3.10.2, each bucket in a weak hashtable is periodically checked for the possibility of being shortened (if the number of reclaimed pointers inside allows it). This check is performed incrementally: a couple of buckets, chosen in a round-robin fashion, are checked each time we need to grow a bucket. Furthermore, the table is resized according to an estimation of its fullness based on the number of buckets with a size above a fixed threshold. Note that the latter would not be an appropriate measure of fullness without the former measure against buckets growing too large.

7.2 Avoid making stored values alive needlessly

In the basic implementation for weak hashtables, the `merge` function, when applied to a value `v`, finds the appropriate bucket and compares `v` to each element it finds in the bucket. This means calling the equality function on each element already in the bucket. The equality function is a perfectly ordinary ML function, which was provided through OCaml's functor system. Like every other ML function, this function expects its arguments to be alive — it would not, in fact, be prepared to handle an argument that wasn't, because the equality function is likely to do some allocations during its execution, which are possibly going to give control to the GC, which is possibly going to finish a cycle and reclaim all the dead blocks it can find. Therefore, it appears necessary to make elements from the bucket alive at the time of passing them to the equality function.

This wouldn't be a problem with a stop-the-world garbage collector, but OCaml's incremental collector uses the snapshot-at-beginning technique, as described in (Wilson 1992). In this setting, making a block alive, even for a short time, means that it cannot be deallocated in the current major GC cycle. In case of frequent access to a weak hashtable, some of its buckets may be traversed at least once during each GC cycle. In this case, their dead blocks will never be deallocated.

In fact, the bucket elements need not be made alive. OCaml's weak arrays allow requesting a shallow copy of the contents of a cell, and in this case, it is the copy that is forced to life, not the original value. The copied block's children, on the other hand, are forced to life in the process, because they are referenced from the copy. These children are the same as the original block's. So, if an element in a weak hashtable is dead and the program is doing a lot of lookups in the table, only the element's top node is guaranteed to be reclaimed on the next GC cycle. The children of the node may have been forced to life by lookups.

This scheme was originally thought to be sufficient... And it is, in a way. Any non-circular dead hashconsed value will be reclaimed after enough GC cycles, even in the worse case where lookups keep making children of the current remaining subtree alive again. One problem with the described scheme is that of circular values stored in weak hashtables. The hashconsing of circular values is an advanced topic but can be encountered (Considine 2000; Mauborgne 2000). The circular hashconsed values may be kept in memory indefinitely by lookups, even when they are otherwise unreachable. Effectively, each lookup for another value that has the same hash will make all the descendants of the circular value alive, including itself. Another problem is more likely to be encountered when using hashconsing techniques: if every node of a tree or DAG is hashconsed, as is for instance the case for Binary Decision Diagrams (Clarke et al. 1999), the deallocation of dead values can be very slow. Indeed, for a now-dead tree that is stored in a weak hashtable, the height of the dead subtrees that keep wasting heap space only going down by one unit at each major GC cycle.

A solution for this issue relies on the fact that most of the time, it is enough to know the hash of each of two values to decide that they are different (if their hashes are different, the values are different). The full hash value of each value stored in a weak hashtable is kept in a cache. Before comparing for equality the value to be `merge`d with a value already present in the same bucket, their hash values are compared. If the full hash values do not match, it is not necessary to call the user-provided `equal` function on the values, and thus the value from the bucket does not need to be made alive. Of course, part of the information contained in the hashes of both values is necessarily identical since their hashes lead them to the same bucket. But it pays to compare the full word of information contained in the values' hashes before calling the `equal` function.

Although it is called less frequently, the internal `resize` function inside the `Weak.Make` functor is subject to a similar remark. This function is called when a weak hashtable is estimated to have become too small for the number of elements it contains. It allocates a new, bigger weak hashtable and copies the elements from the former into the latter. The pre-3.10.2 version of this function called the user-provided `hash` function on each of the values stored in the table, thus forcing the GC to keep them in memory for one additional major cycle, even if they were about to be reclaimed. This function now makes use of the cached hash value for each value in the old table. It avoids calling `hash` and is carefully written so as to avoid forcing to life the referenced values. This implies using a runtime primitive, `Weak.blit`, that was introduced in OCaml 3.10.2 especially for this reason. The primitive `Weak.blit` copies values from a weak array to another weak array without making them alive (using the OCaml functions `Weak.get` and `Weak.set` would make the values alive).

8. A different weak hashtable implementation

The weak hashtables implementation described above with all performance issues fixed is the version distributed with OCaml 3.10.2. Another implementation was made for Frama-C, at a time when it was not clear yet where the issues were in OCaml's provided version. The purpose of this alternative implementation was to lower the memory overhead when storing a large number of elements in a weak hashtable, and especially to lower it so it became possible to have an average of one element per bucket (to emphasize speed) without unreasonable space overhead. At the time, one test case also made it seem like the inherent limitation of OCaml's weak hashtables to 4 million buckets on 32-bit architectures was an issue, so another goal was to raise this limit. It only became clear later that the issue exhibited with this test case was in fact with dead values not being reclaimed quickly enough when they were referred in a weak hashtable, and with improper heuristics for resizing weak hashtables.

The implementation can be found in the Frama-C distribution, in file `buckx.ml` (the name is supposed to evoke hierarchical buckets). This implementation attempts to reduce both wasted space and overhead by mutualizing the space available to several buckets in a single weak array. This implies the management of free cells and of the cells allocated to each bucket very much as if each weak pointer was a block in a filesystem, and each bucket a file. We will call the filesystem-like structure a "meta-bucket" in the rest of this section. At the time of creating a weak hashtable, its size can be chosen by selecting the appropriate number of meta-buckets. To store a value inside a weak hashtable composed of several meta-buckets, part of the value's hash is used to choose a meta-bucket, and the remaining entropy is used to select a bucket inside the meta-bucket.

There is a compromise in the choice of the number of buckets that compose a meta-bucket. An obvious advantage of mutualizing many buckets it that the mutualization is more efficient, whereas with smaller and more numerous meta-buckets, a meta-bucket may still have free cells while another is full and forces the resizing of the table and the redistribution of present elements. There are also less obvious advantages for the choice of mutualizing a smaller number of buckets. One is that the "filesystem-side" overhead has a chance to be smaller when it takes fewer bits to encode the number of a weak cell in the common pool. Another is that when looking for reclaimed pointers in the table, a smaller meta-bucket allows for better locality of the required memory accesses. The implementation settled on using the FAT filesystem (Wikipedia 2008) from 20th-century operating systems for managing one meta-bucket, mutualizing 254 weak pointers (*clusters* in

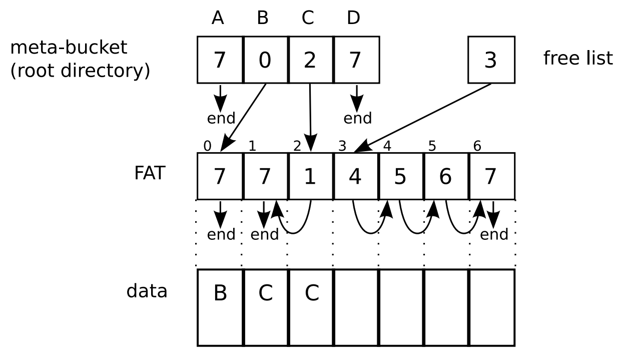

Figure 3. A 3-bit FAT

the FAT analogy) shared between 256 buckets (*root directory entries*). Root directory entries and FAT entries can each be encoded in one byte.

Figure 3 shows an example of a 3-bit FAT. In this figure, each square box uses only 3 bits of memory, while the rectangular ones are regular 32- or 64-bit values. Indexes into the FAT range from 0 to 6, and 7 represents Nil. There are 4 buckets in the meta-bucket (A to D); A and D are empty, B has one entry and C has two entries, linked together via the FAT. The free entries are linked in the meta-bucket's free list.

A separate mechanism limits the ill side-effects of small meta-buckets. It delays the resize operation when some meta-buckets become full before others, by spilling the additional elements in a second smaller structure. When benchmarking this mechanism by storing only live values inside the table, the fill rate would reach between 80% and 90% before a meta-bucket and its spill bucket were both full, forcing to resize the table. Regarding the dead data in the table, another ad-hoc mechanism makes the table more aggressive in its collection of reclaimed weak pointers, at the cost of more CPU usage, when it looks like the available memory or address space of a typical workstation would soon be exhausted.

With the choice of an 8-bit FAT, assuming that the table is full and that the average bucket size is one, the memory occupied by each element of the table is one word for the element itself, one word for caching the hash value of the element, about one byte for the bucket's root directory entry, and a one-byte entry in the FAT, for a total of 10 bytes on a 32-bit architecture. Unevenness in the distribution of elements inside meta-buckets mean that the actual average space occupied per element is between 11 and 12 bytes even just before resizing.

In contrast, if all buckets were of size one in one of OCaml's weak hashtables, the memory occupied per element would be 3 words to store the element (3 words is the heap size occupied by a one-element weak array), 2 words to store the hash value of the element in a normal array, and two words to index both the weak array and the normal array so that they can be accessed. The total is 7 words (28 bytes on a 32-bit architecture). But when the average number of elements per bucket is one, all buckets are unfortunately not of size one with usual hash functions. OCaml's implementation of weak hashtables cleverly shares all buckets of size zero, so the above estimation is wrong. The simulation in appendix A shows that assuming that the hash function is random, more than a third of the buckets can be expected to be empty when 100000 elements are stored in a weak hashtable of 100000 buckets. On the other hand, in order to avoid constant resizing of the buckets, the allocation size for a bucket follows[1] the progression 0, 3, 7, 13, so that for instance, a bucket for 4 elements really contains room for 7. Taking both these biases into account, we arrive to an average size of 3.2 words per element for the weak arrays (and 2.6 words per element for the integer arrays containing the hashes of the elements) when storing 100000 elements in a 100000-buckets weak hashtable, for a total of roughly 31 bytes per element on a 32-bit architecture.

It should be kept in mind, however, that the memory efficiency of OCaml's weak hashtables improves when the average bucket size grows larger than one. Also, the total size of these hashtables grows in a more progressive fashion (each bucket is resized individually), whereas the mutualized buckets must be allocated all at once when resizing one of the weak hashtables described here.

[1] The formula used to compute the new size when re-allocating a bucket is given by the function `next_sz` in file `weak.ml`.

9. Benchmarks

Frama-C (Frama-C development team 2008; Monate and Signoles 2008) is a framework for writing collaborating static analyzers for the C language. One of the existing analyzers inside Frama-C is a value analysis based on abstract interpretation (Cousot and Cousot 1977). This analyzer computes over-approximated (but correct) supersets of the possible values for all variables at all points of the analyzed program. One of the specificities of this analyzer is that it keeps all the information thus computed (the values of all variables in all program points) so as to be able to answer requests from other analyzers when they need it. When using abstract interpretation techniques to detect the possibilities for run-time errors in the analyzed program, only a small fraction of this information needs to be retained at a given time of the analysis. The problem of the increased memory consumption thus caused by the collaborative approach of Frama-C was solved by using Patricia trees (Okasaki and Gill 1998) for the representation of memory states and using hashconsing on these trees. The implementation of big-endian Patricia trees was borrowed from Menhir (Pottier and Régis-Gianas 2005) and heavily adapted.

In order to compare the performance of the various weak hashtable versions available, we launched four different binaries of the Frama-C value analyzer on the C program in appendix B. The compiled source code for the analyzer was identical in each case except for the fact that, in two of the compilations, OCaml's stdlib version of weak hashtables were used and in the other two, Frama-C's weak hashtables were used. Both versions were compiled using both OCaml 3.09.3 and OCaml 3.10.2. The `Weak` module has identical interfaces in these two versions, but OCaml 3.10.2's version of the module `Weak` fixes the problems that are described in this article. The results are reported in the first four lines of table 1.

The analyzer is launched with options that force it to compute as precise a representation as it can of the memory state at the end of the execution[2]. Additionally, after the analysis, for each statement, a (possibly over-approximated) union of the states that have occurred at this statement is available for querying. Intermediate computations consume even more time and memory (the algorithms the analyzer relies on could be modified or improved, but we believe that this does not make this benchmark less relevant as an example of the practical use of hashconsing).

First and foremost, it should be pointed out that these measures are not comparisons of hashconsed implementations versus non-

[2] It is possible to use less time and memory to analyze this program, but then the optimal conclusion concerning the value of `S` (respectively [62475..62975], [499950..500950], [1687425..1688925], [3999900..4001900] for values of `N` being 50, 100, 150, 200) may not be reached. Indeed, the way the program is written, the analyzer needs to have at one point a precise representation of a rather big memory state in order to reach this conclusion.

[3] A bug in /usr/bin/time on the test platform prevents obtaining the maximum resident size when it is larger than 2GiB.

	N = 50	N = 100	N = 150	N = 200
3.09.3 stdlib	78.3s 114MiB	2952s 732 MiB	23582s 1629MiB	101815s >2GiB[3]
3.09.3 buckx	52.2s 98MiB	1149s 349MiB	9427s 654MiB	44714s 991MiB
3.10.2 stdlib	49.1s 85MiB	1018s 326MiB	7936s 624MiB	37472s 936MiB
3.10.2 buckx	50.6s 95MiB	1106s 350MiB	8908s 560MiB	43896s 836MiB
3.10.2 buckx+	50.0s 89MiB	1038s 320MiB	8615s 598MiB	40269s 854MiB

Table 1. User time and maximum RSS as reported by `/usr/bin/time -l` on a 2.66GHz 5150 Intel Xeon Mac Pro with 8GiB of memory (Mac Os X 10.5.2)

hashconsed ones. While unfortunately the non-hashconsed implementation of Frama-C is no longer available for reference, its results would be out of the chart here, with `Out_of_memory` exceptions starting earlier than N = 100 for a 32-bit address space[4]. The improvements by a factor of more than two in time and memory usage that appear in these results are with respect to a hashconsed version of the software that was already much improved compared to the non-hashconsed one.

If we look at the results of table 1 in a little more detail, it is apparent that using either Frama-C's own implementation of weak hashtables or OCaml 3.10.2 makes the analysis significantly leaner and faster than the OCaml 3.09.3 native version of weak hashtables.

Comparing the rows "3.09.3 buckx" and "3.09.3 stdlib" show that Frama-C's own implementation of weak hashtables has played an important role as a stop-gap measure to make Frama-C more efficient when the latest available version of OCaml was 3.09.3. As mentioned previously, the analysis of one C program did exceed the maximum number of buckets a stdlib weak hashtable can have when the analyzer was compiled with OCaml 3.09.3. The advantage it provides is no longer so clear with OCaml 3.10.2. The only difference that remains visible between the two weak hashtables implementations when run on the same 3.10.2 runtime is in the tradeoff between space and CPU usage. The native one is a little faster at the expense of a little additional memory usage.

The differences between the "3.09.3 buckx" and "3.10.2 buckx" versions can be attributed to the improvements in the runtime between the two versions of Ocaml (section 4.3: the addition of `weak_ref_table`). The expected improvement depends on whether the program creates short-lived hashconsed values. Frama-C is in this case, and the run-time improvement is smallish (2-5%) and the memory size improvement is about 15%. On an off-topic note, the problem of the hashconsing of short-lived values is given an interesting twist in (Appel and Gonçalves 1993), although the ideas there do not seem to apply directly to the framework described here.

The wider gap between "3.09.3 stdlib" and "3.10.2 stdlib" takes into account both the improvements in the runtime and in the weak hashtables implementation.

The row "3.10.2 buckx+" in table 1 corresponds to a recent version of the buckx weak hashtables where a performance bug was identified and fixed. Fixing this bug should make the analyzer both faster and more memory efficient. In practice, the bugfix also displaces the equilibrium point the analyzer tries to reach between the total memory used and the time spent collecting garbage (the second ad-hoc mechanism mentioned in section 8). As a result, the time/space ratio seems uninteresting for N=150, value for which the analyzer just entered its memory-saving mode at the end of its analysis without the fix and does not enter it any longer with the fix. For other values of N, the advantage is clearer, although a little more memory is used for N=200 compared to the "3.10.2 buckx" version. The obvious meta-conclusion is that, if the current design for efficient hashconsing in OCaml was considered final, either weak hashtable implementation would probably benefit from being fine-tuned with the help of additional, representative benchmarks.

[4] Frama-C is supported on all the 64-bit architectures that are supported by OCaml, but because of the increased word size, and unless swapping is acceptable to the user, there needs to be at least 6GiB of installed RAM to start improving on the results of the 32-bit version

10. Related work

It is natural to use hashconsing to limit space and time usage when implementing abstract interpretation algorithms, and it seems to be what is done in (Mauborgne 2000) (for the more difficult problem of cyclic values), although the article does not make mention of weak references.

(Filliâtre and Conchon 2006) details a way to implement hashconsing in OCaml (based on the weak arrays provided by the runtime), relying on types to ensure that a non-hashconsed value does not get mistaken with a hashconsed one by accident, and caching the hash value. That article mostly assumes that efficient weak hashtables are available and puts the emphasis on the way to get hashconsing from them, whereas here we describe the construction of efficient weak hashtables in OCaml's flavor. The article uses, as we do, the physical equality == (between hashconsed values, of course). The "type-safe" qualifier in the article's title comes from the fact that the type system will prevent values of the hashconsed type from being created by direct application of the constructors, as opposed to the maximal-sharing-preserving function that applies the constructor tentatively, looks it up in the hashconsing table, and returns the new value only if it was not already in the table. It should be pointed out, however, that this assumes that values can only be created by application of their constructors. The function `Array.get` may create a new value if the value is stored unboxed inside the array (this is already the case with `floats`, and the practice might be generalized in some future version of OCaml). The `Marshal.from_*` functions, *even when used in a type-safe way*, also allocate new values which break the hashconsing invariant. Incidentally, the function `Weak.get_copy` has the same property, and it is not considered type-unsafe. Comparing unique tags instead of using == prevents observing any ill side-effects from the use of `Array.get` or `Weak.get_copy`, although of course the program will still behave incorrectly if it marshals a hashconsed value somewhere, the heap version of the value is later found to be dead, reclaimed by the GC, re-created by application of the constructor, and the program then un-marshals the original, previously marshaled value. Besides, the implementation proposed in (Filliâtre and Conchon 2006) caches the hash of each value as a field inside the type `'a hash_consed`. As we found out (section 7.2), this means by construction that the value will be forced to life when the cached hash value is read. The hash value should ideally be stored separately from the stored value, so that it can be accessed without requiring a call to either `Weak.get` or `Weak.get_copy` on the stored value.

The `WEAK` structure of SML/NJ (Appel and MacQueen 1991) provides a different interface from OCaml, with immutable weak pointers and a small set of primitives, which was used successfully in (Shao 1997). We had to provide a more complex interface that gives more control to the user in order to cope with the difficulties introduced by the incremental GC of OCaml. Moreover, the tradeoffs in OCaml's weak pointers are geared toward hashconsing rather than general use. The primitives provided by SML/NJ make it natural to use a different design for the weak hashtable buckets.

This reflects the influence of the design of weak references on the design of efficient weak hashtables.

11. Future directions

OCaml's weak pointers make it possible to build weak hashtables, but the approach taken here is not ideal. The weak hashtables need to be periodically polled for reclaimed pointers. This is a kind of garbage collection happening within the hashtable, and it is not easy to synchronize with the GC. If the hashtable collections happen too frequently, it is a waste of time because there are not enough reclaimed pointers to make them worthwhile. If they happen too seldom, space is wasted because of the numerous reclaimed pointers using up the room in the table.

It seems clear, as already remarked in (Peyton-Jones et al. 1999) that weak references and finalization should not be considered separately. The solution to the above dilemma is to let the GC take an appropriate bookkeeping action when reclaiming a weak pointer. The nature of the action that would be best for avoiding the need to self-garbage-collect weak hashtables remains to be defined. It is possible to experiment in this direction by using finalization functions to do the bookkeeping but, for performance and scalability, once the appropriate finalization action has been found, it may need to receive special support in the OCaml runtime in order to avoid excessive overhead.

Acknowledgments

The authors are thankful to Benjamin Monate, Virgile Prevosto, Didier Rémy and Julien Signoles for reading early versions of this document and providing useful insights. The anonymous referees' contributions were very valuable even though time was too short to implement all their suggestions before the final revision. Dillon Pariente (Dassault Aviation) provided the example C programs that made it clear that hashconsing was needed in Frama-C.

References

Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Computer Science Department, 1993. URL `citeseer.ist.psu.edu/111544.html`.

Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Proceedings of theThirdInternational Symposium on Programming Language Implementation and Logic Programming*, number 528, pages 1–13. Springer Verlag, 1991. URL `citeseer.ist.psu.edu/appel91standard.html`.

Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. URL `http://citeseer.ist.psu.edu/baker78list.html`.

David Beazley and Guido Van Rossum. *Python; Essential Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 1999. ISBN 0735709017.

Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007. ISBN 978-3-540-75558-6.

Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

Sylvain Conchon and Evelyne Contejean. The Alt-Ergo automatic theorem prover, 2006. URL `http://alt-ergo.lri.fr/`.

Jeffrey Considine. Efficient hash-consing of recursive types, 2000. URL `http://citeseer.ist.psu.edu/article/rey00ecient.html`.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997. URL `http://conal.net/papers/icfp97/`.

Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 12–19, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9. URL `http://doi.acm.org/10.1145/1159876.1159880`.

Frama-C development team. Frama-C: Framework for modular analysis of C, 2008. URL `http://frama-c.cea.fr/`.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201310082.

Eiichi Goto. Monocopy and associative algorithms in an extended lisp. Technical Report TR 74–03, University of Tokyo, 1974.

Bruno Haible. Weak References, Data Types and Implementation, 2005. URL `http://www.haible.de/bruno/papers/cs/weak/WeakDatastructures-writeup.html`.

Xavier Leroy. The Objective Caml system, release 1.07, Documentation and user's manual, 1997. URL `http://caml.inria.fr/pub/distrib/ocaml-1.07/ocaml-1.07-refman.txt`.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, release 3.10, Documentation and user's manual, 2007. URL `http://caml.inria.fr/pub/distrib/ocaml-3.10/ocaml-3.10-refman.txt`.

Zheng Li. Weaktbl, a weak hash table library for OCaml, 2007. `http://www.pps.jussieu.fr/~li/software/weaktbl/README`.

Laurent Mauborgne. Improving the representation of infinite trees to deal with sets of trees. In G. Smolka, editor, *European Symposium on Programming (ESOP 2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 275–289. Springer-Verlag, 2000.

Benjamin Monate and Julien Signoles. Slicing for security of code. In *Trust 2008*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998. URL `http://citeseer.ist.psu.edu/okasaki98fast.html`.

Simon L. Peyton-Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: Weak pointers and stable names in haskell. In *Implementation of Functional Languages*, pages 37–58, 1999. URL `http://citeseer.ist.psu.edu/peytonjones99stretching.html`.

François Pottier and Yann Régis-Gianas. Menhir, December 2005. URL `http://cristal.inria.fr/~fpottier/menhir/`.

Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

Wikipedia. File Allocation Table, 2008. URL `http://en.wikipedia.org/wiki/File_Allocation_Table`.

Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in LNCS, Saint-Malo (France), 1992. Springer-Verlag. URL `http://citeseer.ist.psu.edu/wilson92uniprocessor.html`.

A. Appendix: Simulation

The following is the OCaml code used to evaluate the distribution of the respective sizes of 100000 buckets where 100000 elements have been stored in them.

```
let n = 100000
let t = Array.create n 0
let c = Array.create n 0
  (* continued on next page *)
```

```
let () =
  for i = 0 to n-1 do
    let r = Random.int n in
    t.(r) <- t.(r) + 1
  done;
  for i = 0 to n-1 do
    c.(t.(i)) <- c.(t.(i)) + 1
  done;
  for i = 0 to n-1 do
    Format.printf "%6d -> %6d@\n" i c.(i)
  done
```

Results:

n	number of buckets containing n elements
0	36767
1	36799
2	18406
3	6138
4	1546
5	281
6	57
7	4
8	2

Estimation of the number of memory words used by weak arrays when storing 100000 elements in one of OCaml's weak hashtables:

```
let w = 0 * 36767 +
        5 * (36799 + 18406 + 6138) +
        9 * (1546 + 281 + 57 + 4) +
       15 * 2 ;;
val w : int = 323737
```

Estimation of the number of memory words used by the integer arrays for hashes when storing 100000 elements in one of OCaml's weak hashtables:

```
let h = 0 * 36767 +
        4 * (36799 + 18406 + 6138) +
        8 * (1546 + 281 + 57 + 4) +
       14 * 2 ;;
val h : int = 260504
```

B. Appendix: Benchmark

The following is the C code passed to Frama-C's value analysis for benchmarking various implementations of hashconsing.

```
#define N 150

#include ".../share/builtin.h"
#define FRAMA_C_MALLOC_INDIVIDUAL
#include ".../share/malloc.c"

struct S { int s ; int **t; };

struct S tt[N];

int *P[N];

void init(struct S *ps)
{
  int j,size;
  size = ps->s;
  ps->t[0] = malloc(sizeof(int));
  *(ps->t[0]) = Frama_C_interval(0, 10);
  for (j = 1; j < size; j++)
    {
      ps->t[j] = malloc(sizeof(int));
      *(ps->t[j]) = size + j;
      P[Frama_C_interval(0, j-1)] = ps->t[j];
    }
}

int S;

int sum(struct S *ps)
{
  int j, size, s;
  size = ps->s;
  s = 0;
  for (j = 0; j < size; j++)
    s = s + *(ps->t[j]);
  return s;
}

int main(void)
{
  int i;
  int **p;
  for (i=0; i<N; i++)
    {
      p = (int **) malloc((i+1) * sizeof(int));
      tt[i].s = i+1;
      tt[i].t = p;
      init(&tt[i]);
    }

  for (i=0; i<N; i++)
    {
      S = S + sum(&tt[i]);
    }

  *(P[10]) = -1;

  return S;
}
```

Note that Frama-C does not handle well the analysis of programs where there is non-determinism in the shape of the allocated memory blocks. The calls to `malloc` here are only a convenient way to create large memory states (each `malloc` call is equivalent to creating a fresh variable). In a typical embedded program, there would not be any dynamic allocation but there would be numerous global variables, some of them being complicated structures. The version of Frama-C used for this benchmark is 20080301. The program is analyzed with the following command line:

```
toplevel.opt -val bench.c .../share/builtin.c
   -slevel 210
```

C. For those who kept reading after the credits

The program in appendix B is incorrect: the pointer `P[10]` may not have been initialized when it is accessed at the very end of the program. Frama-C warns about this, but does not emit any other alarm. This means that Frama-C guarantees that this line is the only one where there might be an invalid memory access in this program.