Hash consing in an ML framework

Jean-Christophe Filliâtre

LRI, Université Paris-Sud, France
filliatr@lri.fr

Abstract

Hash consing is a technique to share values that are structurally equal. Beyond the obvious advantage of saving memory blocks, hash consing may also be used to gain speed in several operations (like equality test) and data structures (like sets or maps) when sharing is maximal. However, physical adresses cannot be used directly for this purpose when the garbage collector is likely to move blocks underneath. We present an easy solution in such a framework, with many practical benefits.

1 Introduction

Hash consing is a technique to share purely functional data that are structurally equal [4]. The name "hash consing" comes from Lisp: the only allocating function is *cons* and sharing is traditionally realized using a hash table [2]. One obvious use of hash consing is to save memory space. Extreme approaches even suggest ML runtimes where hash consing is performed in a systematic way by the garbage collector [3, 5]. Yet, there is an overhead in looking for already allocated values and this can result in a loss of performance.

When sharing is maximal, however, hash consing may also result in a gain of performance. Indeed, physical equality (written `==` in this paper) can be substituted for structural equality (written `=`) since we have now the following property

$$x = y \quad \Rightarrow \quad x == y$$

(Only the converse is always true in general.) Then, equality testing is now a constant time operation, instead of requiring a time proportional to the size of the data. The property of maximal sharing can also be used to build efficient sets and maps, using for instance binary tries based on the pointers' digits [5]. However, this cannot be done so easily when the garbage collector is likely to move blocks.

In this paper, we present an implementation of hash consing in such a framework, with many practical benefits. We will use the syntax of Objective Caml (Ocaml for short) [1] when giving pieces of code, but this can be translated to SML very easily.

This paper is organized as follows. Section 2 introduces some notations and our assumptions on hashing. Section 3 quickly presents an obvious implementation of hash consing and discusses its drawbacks. Then Section 4 presents our solution to these drawbacks. Finally, Section 5 makes it even more efficient by introducing a hash table specialized in hash consing.

2 Hashing

In the following, we assume that we have a type `t` equipped with an equality and a hash function *i.e.* we have a module `H` implementing the following signature:

```
module type HashedType = sig
  type t
  val equal: t × t → bool
  val hash: t → int
end
```

The hashing function is supposed to be consistent with the equality function *i.e.* that we have (`hash` x) = (`hash` y) as soon as `equal` (x, y) holds. Then we get an implementation of hash tables over type `H.t` by instantiating the functor provided in Ocaml's standard library module `Hashtbl`:

```
module HashTable = Hashtbl.Make(H)
```

The resulting module `HashTable` implements the following signature[1]:

```
sig
  type α t
  val create : int → α t
  val add : α t → H.t → α → unit
  val find : α t → H.t → α
end
```

where α `t` is the type of hash tables mapping elements of type `H.t` to values of type α; `create` returns a new hash table with a given initial size; `add` inserts a new binding in a table, hiding any previous binding for the same key; and `find` looks for the value associated to a given key, raising the predefined exception `Not_found` if there is no such binding.

As a running example of a data-type on which to perform hash consing, we choose the following type `term` for λ-terms with de Bruijn indices:

```
type term =
  | Var of int
  | Lam of term
  | App of term × term
```

We can equip this type with Ocaml's polymorphic equality and hash function to get the following implementation of module `H`:

```
module H = struct
  type t = term
  let equal (x,y) = (x = y)
  let hash = Hashtbl.hash
end
```

[1]Other functions are provided by the module `Hashtbl`, but they are not relevant in this paper.

3 Simple hash consing

The standard way of doing hash consing is to use a global hash table to store already allocated values and to look for an existing equal value in this table every time we want to create a new value. It leads to the following code:

```
let hashcons_term =
  let table = HashTable.create 251 in
  fun x →
    try HashTable.find table x
    with Not_found → HashTable.add table x x; x
```

The initial size of the hash table is clearly context dependent, and we will not discuss its choice in this paper. (Anyway, choosing a prime number is always a good idea.) Here, the resulting hash consing function has the following type:

```
val hashcons_term : term → term
```

If we want to get *maximal sharing*—the property that two values are indeed shared as soon as they are structurally equal—it is a good idea to introduce new "constructors" performing hash consing:

```
let var n = hashcons_term (Var n)
let lam u = hashcons_term (Lam u)
let app (u,v) = hashcons_term (App (u,v))
```

Therefore, by applying `var`, `lam` and `app` instead of `Var`, `Lam` and `App` directly, we ensure that all the values of type `term` are always hash consed. When maximal sharing is achieved, physical equality (`==`) can be substituted for structural equality (`=`) since we have now

$$x = y \quad \Leftrightarrow \quad x == y$$

In particular, the equality used in the hash consing itself can now be improved by using physical equality on sub-terms, since they are already hash consed by assumption. In practice, we define the following function

```
let eq_sub_term = function
  | Var n, Var m → n == m
  | Lam u, Lam v → u == v
  | App (u1,u2), App (v1,v2) → u1 == v1 & u2 == v2
  | _ → false
```

right after the definition of type `term` and right before the implementation of module `H`, which is now the following:

```
module H = struct
  type t = term
  let equal = eq_sub_term
  let hash = Hashtbl.hash
end
```

Drawbacks. Given a good hash function, the above implementation of hash consing is already quite efficient. But it still has the following drawbacks:

- There is no way to distinguish between the values that are hash consed and those which are not, and the programmer has to take care that the true constructors are never used directly without the surrounding call to the hash consing function. Of course, the type could be made abstract but then we would lose the pattern-matching facility.

- Maximal sharing allows us to use physical equality instead of structural equality, which can yield substantial speedups. If one would have access to the physical address of ML values, one could even think of using it to build efficient data structures. It is done by J. Goubault in [5] for sets and maps, using binary tries based on the binary digits of the addresses. However, this cannot be done when the garbage collector is likely to move allocated blocks underneath, which is the case of Ocaml's collector for instance. And we do not have access to the address anyway.

- Our implementation of the hash consing function computes twice the hash value associated to its argument when it is not in the hash table: once to look it up and once to add it. It is also wasting space in the buckets of the hash table by storing twice the pointer to `x`. These two drawbacks could be fixed by designing an implementation of hash tables specialized in hash consing.

In the following, we present solutions to these problems. Section 4 addresses the first two and Section 5 addresses the last one.

4 Tagging the hash consed values

The idea is to tag the hash consed values with unique integers. First, it introduces a type distinction between the values that are hash consed and those that are not. Second, these tags can be used to compare hash consed values, in order to build data structures requiring ordering, like balanced trees. For the purpose of tagging, we introduce the following record type:

```
type α hash_consed = {
  node : α;
  tag : int }
```

The field `node` contains the value itself and the field `tag` the unique tag. Of course, the introduction of this new type implies a change in the definition of types for values to be hash consed. In our example, the definition of type `term` becomes

```
type term_node =
  | Var of int
  | Lam of term
  | App of term × term

and term = term_node hash_consed
```

All the nodes of type `term` are now decorated with tags. The definition of `eq_sub_term` is still the same, but it is now an equality over type `term_node`. Therefore, the implementation of module `H` becomes

```
module H : HashedType = struct
  type t = term_node
  let equal = eq_sub_term
  let hash = Hashtbl.hash
end
module HashTable = Hashtbl.Make(H)
```

Then we can rewrite the function `hashcons`, which is now of type `term_node` → `term`. It uses an internal counter to associate a new tag to each value that is not in the table.

```
let hashcons_term =
  let table = HashTable.create 251 in
  let gen_tag = ref 0 in
  fun x →
    try
      HashTable.find table x
    with Not_found →
      let hx = { node = x; tag = !gen_tag } in
      incr gen_tag;
      HashTable.add table x hx;
      hx
```

The hash consing constructors `var`, `lam` and `app` are still defined in the same way and still have the expected types, that are:

```
val var : int → term
val lam : term → term
val app : term × term → term
```

Maximal sharing is now easily enforced by type checking, since a direct use of the true constructors would build values of type `term_node` instead of type `term`. We now have the following property:

$$x \texttt{ = } y \quad\Leftrightarrow\quad x \texttt{ == } y \quad\Leftrightarrow\quad x\texttt{.tag = }y\texttt{.tag}$$

In particular, tags can be used to build comparison functions over hash consed values. In Ocaml, functors requiring comparison functions expect a single function of type $\alpha \to \alpha \to$ `int`, where the sign of the returned integer is mapped to the three possible relations $<$, $=$ and $>$. In our example, we can introduce the following comparison over terms:

```
let compare_term x y = x.tag - y.tag
```

Then we can build sets of terms and maps over terms using the modules `Set` and `Map` from Ocaml's standard library in the following way:

```
module OrderedTerm = struct
  type t = term
  let compare = compare_term
end
module TermSet = Set.Make(OrderedTerm)
module TermMap = Map.Make(OrderedTerm)
```

The modules Set and Map are implemented using balanced trees with fairly good performance. But since elements (resp. keys) are actually integers, we can build more efficient data structures. Using binary tries based on bits as done in [5] is already a good idea, but Patricia trees is an even better solution, as recently presented in an ML framework by C. Okasaki and A. Gill [6].

5 A specialized hash table

As explained at the end of Section 3, our implementation of the hash consing function is not very efficient for two reasons:

- First, it wastes space by storing twice the same value in the hash table buckets;
- Second, it wastes time by computing twice the hash key of the given value, once to look for it and once to insert it.

These two problems have an easy solution, that is to design a hash table specialized in hash consing. The interface of such a specialized module is the following:

```
module type S = sig
  type key
  type t
  val create : int → t
  val hashcons : t → key → key hash_consed
end
```

and its implementation is a functor having the following type:

```
module Make(H : HashedType) : (S with type key = H.t)
```

The implementation of this functor is immediate: a hash consing table is an array of lists containing hash consed values (*i.e.* of type `H.t hash_consed`). Thus we define

```
type key = H.t
type t = key hash_consed list array
```

The creation of an empty hash consing table is just the creation of an array of empty lists:

```
let create n = Array.create n []
```

Given a hash consing table `t`, hash consing a value `v` is performed by calling `hashcons t v`, where `hashcons` is defined in the following way:

```
let gen_tag = ref 0

let hashcons t v =
  let i = (H.hash v) mod (Array.length t) in
  let rec look_and_add = function
    | [] →
        let hv = { tag = !gen_tag; node = v } in
        t.(i) ← hv :: t.(i);
        incr gen_tag;
        hv
    | hd :: tl →
        if H.equal (hd.node,v) then hd else look_and_add tl
  in
  look_and_add t.(i)
```

First, we compute the hash index of `v`, in the variable `i`. Then we traverse the list `t.(i)` looking for an already hash consed value `hd` equal to `v`. If we find such a value we just return it. Otherwise we reach the end of the list and we create a new hash consed value `hv` that we add to the list `t.(i)`. Then we increment the counter `gen_tag` and we return `hv`.

In practice, our implementation of hash consing tables also provides functions to clear tables and to iterate over the entries, but this is not relevant in this paper. We also started from a more efficient implementation of hash tables, where the array is enlarged when the number of collisions becomes too important. The hash keys are stored in the buckets to avoid their recomputations when the table is resized[2].

Another optimization is to define the hashing function `H.hash` using the tags of the sub-terms—remember that these sub-terms are already hash consed. It is particularly efficient since it avoids traversing the terms to compute their hash keys and since the tags are, by definition, distinct integers.

6 Conclusion

We have presented an implementation of hash consing in ML with many practical benefits. In particular, it introduces a type distinction between hash consed values and normal values, that statically enforces invariants of the code. It also allows the construction of efficient set and maps of hash consed values, even in contexts where blocks can be moved by the garbage collector.

In practice, we applied this hash consing implementation in the context of a state of the art decision procedure, where structural equality between terms and maps over terms were heavily used. We got a substantial speedup (between 2 and 3, depending on the nature of inputs).

However, our implementation has the slight drawback of using extra space to store the tags associated to the hash consed values. On the other hand, hash consing is also saving

[2] Ocaml's standard hash tables do not use that optimization but SML ones do.

space by sharing. And it also trades space for time, by allowing a more efficient implementation of some operations. Therefore, the pertinence of hash consing can be discussed depending on the application.

Another drawback concerns garbage collection: once a value is hash consed, it is alive for ever as far as garbage collection is concerned because it persists in the hash consing table, unless we clear or throw away the table at some point. This issue has a solution when hash consing is performed by the garbage collector itself, as done in [3] or [5].

The implementation of the module `Hashcons` presented in Section 5 is freely available at `http://www.lri.fr/~filliatr/software.en.html`, together with implementations of sets and maps using Patricia trees following [6].

References

[1] The Objective Caml language. `http://caml.inria.fr/`.

[2] John Allen. *Anatomy of Lisp.* McGraw-Hill Book Compagny, 1978.

[3] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing Garbage Collection. Technical Report CS–TR–412–93, Princeton University, February 1993.

[4] Eiichi Goto. Monocopy and associative algorithms in extended Lisp. Technical Report TR 74–03, University of Tokyo, May 1974.

[5] Jean Goubault. Implementing Functional Languages with Fast Equality, Sets and Maps: an Exercise in Hash Consing. In *Journées Francophones des Langages Applicatifs (JFLA'93)*, pages 222–238, Annecy, February 1993.

[6] Chris Okasaki and Andrew Gill. Fast Mergeable Integer Maps. In *Workshop on ML*, pages 77–86, September 1998.