Type-Safe Modular Hash-Consing

Jean-Christophe Filliâtre
LRI
Université Paris Sud 91405 Orsay France
filliatr@lri.fr

Sylvain Conchon
LRI
Université Paris Sud 91405 Orsay France
conchon@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 speed up fundamental operations and data structures by several orders of magnitude when sharing is maximal. This paper introduces an OCAML hash-consing library that encapsulates hash-consed terms in an abstract datatype, thus safely ensuring maximal sharing. This library is also parameterized by an equality that allows the user to identify terms according to an arbitrary equivalence relation.

Categories and Subject Descriptors D.2.3 [*Software engineering*]: Coding Tools and Techniques

General Terms Design, Performance

Keywords Hash-consing, sharing, data structures

1. Introduction

Hash-consing is a technique to share purely functional data that are structurally equal [8, 9]. 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.

Hash-consing is part of the programming folklore but, in most programming languages, it is more a design pattern than a library. 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. For instance, in the Objective Caml programming language[1] it reduces to the following four lines of code using hash tables from the OCAML standard library:

```
let table = Hashtbl.create 251
let hashcons x =
  try Hashtbl.find table x
  with Not_found → Hashtbl.add table x x; x
```

The `Hashtbl` module uses the polymorphic structural equality and a generic hash function. 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.

As a running example of a datatype 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
```

Instantiated on this type, the `hashcons` function has the following signature:

```
val hashcons : term → term
```

If we want to get *maximal sharing*—the property that two values are indeed shared as soon as they are structurally equal—we need to systematically apply `hashcons` each time we build a new term. Therefore it is a good idea to introduce *smart constructors* performing hash-consing:

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

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. Thus maximal sharing is achieved and physical equality (`==`) can be substituted for structural equality (`=`) since we now have

$$x = y \iff x == y.$$

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. To do such a bootstrapping, we need custom hash tables based on this new equality. Fortunately, the OCAML standard library provides generic hash tables parameterized by arbitrary equality and hash function. To get custom hash tables, we simply need to define a module that packs together the type `term`, an equality and a hash function

```
module Term = struct
  type t = term
  let equal x y = match x,y with
    | Var n, Var m → n == m
    | Lam u, Lam v → u == v
    | App (u1,u2), App (v1,v2) →
        u1 == v1 && u2 == v2
    | _ → false
  let hash = Hashtbl.hash
end
```

and then to apply the `Hashtbl.Make` functor:

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

[1] We use the syntax of Objective Caml (OCAML for short) [1] throughout this paper, but this could be easily translated to any ML implementation.

ML'06 September 16, 2006, Portland, Oregon, USA.

Finally, we can redefine our hash-consing function in order to use this improved implementation:

```
let table = H.create 251
let hashcons x =
  try H.find table x
  with Not_found → H.add table x x; x
```

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

1. There is no way to distinguish between the values that are hash-consed and those which are not. Thus the programmer has to carefully use the smart constructors only and never the true constructors directly. Of course, the type `term` could be made abstract (but then we would lose the pattern-matching facility) or better a private datatype (where pattern-matching is allowed but direct construction is not).
2. Maximal sharing allows us to use physical equality instead of structural equality, which can yield substantial speedups. However, we would also like to improve data structures containing hash-consed terms, such as sets or dictionaries implemented using hash tables or balanced trees for instance. We could think of using the pointers to get $O(1)$ hashing or total ordering functions over hash-consed terms (as done in [10]) but this is not possible for two reasons. First, the OCAML compiler does not give access to the pointer value. Second, the OCAML garbage collector is likely to move allocated blocks underneath, thus changing the values of pointers.
3. There is no possibility for the garbage collector to reclaim terms that are not alive anymore (from the user point of view) since they are referenced in the hash-consing table. If this table is defined at toplevel (and it usually is), the hash-consed terms will never be collected.
4. 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[2]. It is also wasting space in the buckets of the hash table by storing twice the pointer to `x`.

In this paper, we present an OCAML hash-consing library addressing all these issues. Section 2 introduces the design and the implementation of this library. Then Section 3 presents several case studies and reports on the time and space performance effects of hash-consing.

2. A hash-consing library

This section presents an OCAML hash-consing library. We first detail its design and interface, then its practical use and finally its implementation.

2.1 Design and interface

The main 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 build efficient data structures over hash-consed values, such as hash tables or balanced trees. For the purpose of tagging, we introduce the following record type:

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

The field `node` contains the value itself and the field `tag` the unique integer. The `private` modifier prevents the user from building values of type `hash_consed` manually, yet allowing field access and pattern matching over this type. The field `hkey` stores the hash key of the hash-consed term; it is justified in the implementation section below.

The library consists in a functor `Make` taking a type equipped with equality and hashing functions as argument and returning a data structure for hash-consing tables. The input signature is the following:

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

with the implicit assumption that the hashing function is consistent with the equality function *i.e.* `hash` x = `hash` y as soon as `equal` (x, y) holds. Then the functor looks like:

```
module Make(H : HashedType) : sig
  type t
  val create : int → t
  val hashcons : t → H.t → H.t hash_consed
end
```

The datatype `t` for hash-consing tables is abstract. `create` n builds a new hash-consing table with initial size n. As for traditional hash tables, this size is somewhat arbitrary (the table will be resized when necessary). Then, if h is a hash-consing table, `hashcons` h t builds the hash-consed value for a given term t.

In practice, there is also a function `iter` to iterate over all the terms contained in a hash-consing table and a function `stats` to get statistics on a table (number of entries, biggest bucket length, etc.). There is also a non-functorial version of the library, based on structural equality and the generic OCAML hash function, but this is clearly less useful (these two functions do not exploit the sharing of sub-terms).

Due to the private type `hash_consed`, maximal sharing is now easily enforced by type checking (as long as we use a single hash-consing table, of course). As a consequence, the following holds:

$$x = y \iff x == y \iff x.\texttt{tag} = y.\texttt{tag} \qquad (1)$$

2.2 Usage

Before entering the implementation details of the hash-consing library, let us demonstrate its use on our running example. First, we need to adapt our datatype for λ-terms to interleave the `hash_consed` nodes with our original definition:

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

It is important to notice that this modification is not intrusive at all (from the syntactic point of view) and applies to mutually recursive types as well. Indeed, the definition of a set of types such as

$$\begin{aligned}\texttt{type}\ t_1 &= \langle\text{definition}_1\rangle \\ &\vdots \\ \texttt{and}\ t_n &= \langle\text{definition}_n\rangle\end{aligned}$$

[2] On top of that, OCAML's hash tables also waste time recomputing the hash values when resizing the hash tables, contrary to SML's hash tables where hash keys are stored in the buckets. We also address this issue in the next section.

simply needs to be turned into

$$\texttt{type } t_1 \;= t_1\texttt{\_node hash\_consed and } t_1\texttt{\_node } = \langle\text{definition}_1\rangle$$
$$\vdots$$
$$\texttt{and } t_n \;= t_n\texttt{\_node hash\_consed and } t_n\texttt{\_node } = \langle\text{definition}_n\rangle$$

and we notice that the type definitions $\langle\text{definition}_i\rangle$ are left unchanged.

Then we can equip the type `term_node` with suitable equality and hashing function in order to apply the hash-consing functor. As we did in Section 1, equality may safely use physical equality on sub-terms and thus runs in $O(1)$. The hashing function can also be improved to run in constant time by making use of the hash keys for the sub-terms (the `hkey` field) or equivalently their tags (the `tag` field). Altogether, we get the following module definition

```
module Term_node = struct
  type t = term_node
  let equal t1 t2 = match t1, t2 with
    | Var i, Var j → i == j
    | Lam u, Lam v → u == v
    | App (u1,v1), App (u2,v2) →
        u1 == u2 && v1 == v2
    | _ → false
  let hash = function
    | Var i → i
    | Lam t → abs (19 × t.hkey + 1)
    | App (u,v) →
        abs (19 × (19 × u.hkey + v.hkey) + 2)
end
```

We get an implementation of hash-consing tables for this type via a simple functor application:

```
module Hterm = Hashcons.Make(Term_node)
```

Then we can define a global hash-consing table and *smart constructors* for `Var`, `Lam` and `App` performing hash-consing, as follows:

```
let ht = Hterm.create 251
let var n = Hterm.hashcons ht (Var n)
let lam u = Hterm.hashcons ht (Lam u)
let app (u,v) = Hterm.hashcons ht (App (u,v))
```

These "constructors" have the expected types, namely

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

and thus, from this point, the user can simply ignore the hash-consing mechanics that is performed underneath when building terms.

We can now exploit the unique tags to build efficient data structures over terms. For instance, these tags define a total ordering over terms and we can use it to build balanced binary trees containing terms. Using functors from OCAML's standard library, we get implementations of sets of terms and dictionaries indexed by terms as follows[3]:

```
module OrderedTerm = struct
  type t = term
  let compare x y =
    Pervasives.compare x.tag y.tag
end
module TermSet = Set.Make(OrderedTerm)
module TermMap = Map.Make(OrderedTerm)
```

[3] `Pervasives.compare` is to be preferred to a subtraction due to possible integer overflows.

The modules `Set` and `Map` have fairly good performances. But since elements are actually (tagged with) integers, we can build even more efficient data structures such as Patricia trees [12]. Our library comes with Patricia trees-based sets and dictionaries specialized for hash-consed values.

Similarly, we could build efficient hash tables indexed by terms, using the field `hkey` (or even the tag) as an immediate hash value.

2.3 Implementation

Up to now, we have solved issues 1 and 2 that were listed Section 1 (a distinct type for hash-consed values and efficient data structures based on physical equality). We now address issues 3 and 4.

Regarding time and space inefficiency related to the use of hash tables from OCAML's standard library, the first improvement is to build a custom hash table where buckets are simply *lists* of values, and not mapping of values to themselves, saving one pointer for each value. Then the next improvement is to write a single lookup-or-insert function that computes the hash key only once. Finally, the last improvement is to record the hash key (in the field `hkey` which is exposed to the user) to avoid recomputing it when resizing the table (if required).

Regarding the ability for the garbage collector to reclaim the hash-consed values that are not referenced anymore (from anywhere else than the hash-consing table), the solution is to use *weak pointers*. A weak pointer is precisely a reference that does not prevent the garbage collector from erasing the value it points to. Said otherwise, a memory cell may be collected as soon as it is referenced only by weak pointers. The OCAML standard library provides arrays of weak pointers [1] where the access operation may return either `Some` x when there is some available element x and `None` otherwise (which means that the element has been reclaimed by the GC at some point in the past).

Combining the ideas of a custom hash table and the use of weak pointers, we get a *weak hash table* [4], that is a hash table where the buckets are arrays of weak pointers. The beginning of our hash-consing functor is thus as follows:

```
module Make(H : HashedType) = struct

  type t = {
    table : H.t hash_consed Weak.t array;
    ...
  }
```

We omit the other fields that contain irrelevant data related to the resizing heuristics. The insertion and resizing code is standard and we give the main ideas only. The sole difference with respect to OCAML weak hash tables is that we do no need to (re)compute the hash key when inserting or resizing since it is contained in the data itself.

```
let rec resize t =
  ... increase the size of the main array
      and redistribute the elements in the
      new buckets

and add t d =
  let index =
    d.hkey mod (Array.length t.table)
  in
  ... lookup for an empty slot in the
      bucket t.data.(index)
  ... if found then insert d
      else increase the bucket and insert d
```

[4] The OCAML standard library also provides weak hash tables and we borrowed most code from this implementation.

```
    ... if the limit is reached then resize t
```

Then the main `hashcons` operation consists in a single lookup-or-insert function. The pseudo-code is as follows:

```
  let hashcons t d =
    let hkey = H.hash d in
    let index =
      hkey mod (Array.length t.table)
    in
    ... lookup in the bucket t.data.(index)
        for a value v such that H.equal v.node d
    ... if found then return v else
    ... let n = { hkey = hkey; tag = newtag ();
                  node = d } in
        add t n;
        n
```

where `newtag` is a function returning distinct integers. It may seem rather inefficient to first scan the bucket for an equal value already hash-consed and then, in case of a failure, to scan it again for an empty slot to insert it. We could indeed remember any empty slot encountered while scanning for an equal value and then use it for insertion, if any. But in practice this is not worth maintaining this information. Indeed, the buckets are quite small (we start with 3 elements buckets and add only 3 new slots each time we increase them).

3. Case Studies

This section demonstrates the benefits of hash-consing on several case studies.

3.1 Reducing λ-terms

This first case study uses our running example to perform a little benchmark that is representative of massive symbolic computations, as arising for instance in proof assistants. The benchmark is inspired by Huet's *Constructive Computation Theory* [11]. It consists in running a pure λ-calculus version of the quicksort algorithm on a λ-term encoding a list of integers, themselves represented using Church's numerals encoding (see [11], Section 2.2.1, pages 26–28). The datatype for λ-terms is the one we have already used as our running example. Only a very small code excerpt from [11] is needed to perform the computation, namely the four functions `lift` (lifting), `subst` (substitution), `hnf` (head normal form) and `nf` (normal form), all of them totalizing 31 lines of code. The list that is sorted is the 6 element list `[0;3;5;2;4;1]`. It may seem rather small but it already involves 1.6 million elementary substitutions (due to an exponential complexity).

The code was compiled with the OCAML native-code compiler (`ocamlopt`) on a Pentium 4 processor running under Linux. The timings are given in seconds and correspond to CPU time. The memory use corresponds to the maximum size reached by the major heap, in kilobytes, as reported by the `Gc.stat` library function. Each program is run five times in a row and the median of the last three runs is reported.

We first run the computation with and without hash-consing. The results are the following:

	without hash-consing	with hash-consing
time	91.5 s	195 s
memory use	1,680 kb	480 kb

As expected, the use of hash-consing greatly reduces the amount of memory used (more than 3 times less) and this has a cost (the program is more than 2 times slower). Indeed, much time is spent in lookups while terms are created. It is a usual tradeoff between time and space. Though it may seem disappointing, the best is to come.

Then we add *memoization* (also know as *dynamic programming*): each of the four functions `lift`, `subst`, `hnf` and `nf` records its results in a hash table to avoid performing the same computation twice. For the code without hash-consing, we use traditional hash tables based on structural equality and a structural hash function. But for the code where terms are hash-consed we use hash tables indexed by the integers tags. The results are the following:

	without hash-consing	with hash-consing
time	54 s	5.06 s
memory use	95,300 kb	720 kb

Now the version with hash-consing is much faster than its counterpart (more than 10 times faster) but it is also using much less memory. When compared to the original version without hash-consing nor memoization, the final code with both features enabled performs much better regarding time *and* space. This example is typical of the use of hash-consing to improve not only the memory use but, above all, the efficiency of data structures involving hash-consed terms.

$$\textsc{Assume}\ \frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, l}$$

$$\textsc{Bcp}\ \left\{ \begin{array}{c} \dfrac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, \bar{l} \vee C} \\[2ex] \dfrac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, l \vee C} \end{array} \right.$$

$$\textsc{Unsat}\ \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}$$

Figure 1. Abstract version of DPLL

3.2 Boolean Satisfiability Checking

We show in this section how the hash-consing library presented so far can be used to write an efficient SAT-solver.

3.2.1 SAT-solvers

SAT-solvers decide the satisfiability of propositional formulas. Traditionally, they are based on the DPLL procedure [7, 6] for which we give an abstract version Figure 1. The state of the procedure is represented by a sequent $\Gamma \vdash \Delta$ where Γ is a set of literals (propositional variables or their negations) and Δ is a set of clauses (disjunctions of literals).

A propositional formula F is satisfiable if and only if a sequent $\Gamma \vdash \emptyset$ can be derived from the initial sequent $\emptyset \vdash \Delta$ where Δ is the conjunctive normal form (CNF) of F.

Putting a propositional formula in CNF may cause an exponential blow-up in size. We show in the next section how the use of our hash-consing library allows us to solve elegantly the CNF conversion problem. Then Section 3.2.3 shows the implementation of a SAT-solver on top of this CNF conversion.

3.2.2 Equisatisfiable CNF

A well-known solution to the CNF blow-up consists in introducing *proxy-variables* (proxies for short) for all subformulas of the input formula F and to initialize the solver with all the clauses defining these proxies plus the proxy variable for F. The set of clauses thus obtained is *equisatisfiable* to the original formula.

For instance, introducing a proxy X for a sub-formula $P \wedge Q$ (where P and Q are propositional variables) amounts to introducing the three following *proxy-clauses* (pclauses for short)

$$\begin{array}{l}\neg X \vee P\\ \neg X \vee Q\\ X \vee \bar{P} \vee \bar{Q}\end{array}$$

whose conjuction is equivalent to $X \leftrightarrow (P \wedge Q)$. Now, in order to restrict further the number of clauses manipulated by the solver, these pclauses can be incrementally introduced in the context of the solver at the moment where the proxy X or its negation $\neg X$ are asserted. In that case, it is obvious that only the clause $\bar{P} \vee \bar{Q}$ has to be introduced when $\neg X$ is assumed and only P and Q when X is asserted. The following table shows the pclauses introduced by the solver when it asserts a proxy X (or its negation) representing formulas made with standard boolean connectives (P is a propositional variable, X, Y and Z are proxies).

Proxy	X asserted	$\neg X$ asserted
$X \leftrightarrow P$	$\{P\}$	$\{\bar{P}\}$
$X \leftrightarrow Y \wedge Z$	$\{Y\}\,\{Z\}$	$\{\neg Y \vee \neg Z\}$
$X \leftrightarrow Y \vee Z$	$\{Y \vee Z\}$	$\{\neg Y\}\,\{\neg Z\}$
$X \leftrightarrow (Y \rightarrow Z)$	$\{\neg Y \vee Z\}$	$\{Y\}\,\{\neg Z\}$

We notice that a pclause is either a disjunction of two proxies, a set of two unit clauses (each of them containing a proxy), or a single unit clause (with a propositional variable). From that remark, we follow the same steps as in Section 2.2 to define the following types for representing equisatisfiable CNF formulas.

```
type pclause =
    C of t×t | U of t×t | L of string×bool
and view = { pos : pclause; neg : pclause}
and t = view Hashcons.hash_consed
```

Hash-consed values of type `t` represent proxies. The type `pclause` is the type of pclauses. Its constructor `C` stands for pclauses made of two proxies, `U` represents the conjunction of two unit clauses containing proxies and `L` is used for building unit clauses containing literals. Record values of type `view` contain the pclauses that should be added when asserting (positively or negatively) a proxy.

In order to improve sharing, we define an equality relation (and a suitable hashing function) that attempts to identify some subformulas that are logically equivalent. For instance, proxies for $X \rightarrow Y$ and $\neg X \vee Y$ would refer to the same variable. We get an implementation of hash-consing tables by applying the `Hashcons.Make` functor to the following module:

```
module View = struct
  open Hashcons
  type t = view
  let eqc c1 c2 = match c1,c2 with
      U(f1,f2) , U(g1,g2)
    | C(f1,f2) , C(g1,g2) →
       f1==g1 && f2==g2 || f1==g2 && f2==g1
    | L(x1,b1) , L(x2,b2) → x1=x2 && b1=b2
    | _ → false
  let equal f1 f2 =
     eqc f1.pos f2.pos && eqc f1.neg f2.neg
  let hashc acc = function
      U(f1,f2) | C(f1,f2) →
        let min = min f1.tag f2.tag in
        let max = max f1.tag f2.tag in
        (acc×19 + min)×19 + max
    | L _ as z → Hashtbl.hash z
  let hash f = abs (hashc (hashc 1 f.pos) f.neg)
end
module H = Hashcons.Make(View)
```

Then, we introduce a global table to store the equisatisfiable CNF and the corresponding smart constructors.

```
open Hashcons
let tbl = H.create 251
let view t = t.node
let compare f1 f2 = compare f1.tag f2.tag
let equal f1 f2 = f1.tag == f2.tag

let mk_atom a =
  H.hashcons tbl ({pos=L(a,true);neg=L(a,false)})
let mk_not f = let f = view f in
   H.hashcons tbl ({pos=f.neg;neg=f.pos})
let mk_and f1 f2 = if equal f1 f2 then f1 else
  H.hashcons tbl
    {pos=U(f1,f2); neg=C(mk_not f1,mk_not f2)}
let mk_or f1 f2 = ...
let mk_imp f1 f2 = ...
```

A slight improvement is made here so that the proxies returned for $X \wedge X$ and $X \vee X$ are both X.

It is then easy to define a function `cnf : formula → t` that computes the CNF conversion of a propositional formula by recursively applying the smart constructors `mk_and`, `mk_imp`, ...

3.2.3 Implementation

We now assume that the code presented in the previous section is in a module `Cnf`. The state of our SAT solver is represented by the set of proxies that have been assumed and a list of pair of proxies:

```
module S = Set.Make(Cnf)
type t = { gamma : S.t ;
           delta : (Cnf.t×Cnf.t) list}
```

The core of the solver is a pair of two mutually recursive functions `assume` and `bcp`. The former assumes a proxy and thus adds the corresponding pclauses to the context. The latter performs boolean constraint propagation.

```
exception Sat
exception Unsat

let rec assume env f =
  if S.mem (Cnf.mk_not f) env.gamma then
    raise Unsat;
  if S.mem f env.gamma then env
  else
    let env =
      { env with gamma = S.add f env.gamma }
    in
    match Cnf.view f with
      Cnf.Proxy {Cnf.pos=Cnf.U(f1,f2)} →
        assume (assume env f1) f2
    | Cnf.Proxy {Cnf.pos=Cnf.C(f1,f2)} →
        bcp { env with
              delta=(f1,f2)::env.delta }
    | _ → bcp env

and bcp env =
  let cl , u =
    List.fold_left
      (fun (cl,u) (f1,f2) →
        if S.mem f1 env.gamma
```

```
        || S.mem f2 env.gamma
           then (cl,u)
        else if S.mem (Cnf.mk_not f1) env.gamma
           then (cl,f2::u)
        else if S.mem (Cnf.mk_not f2) env.gamma
           then (cl,f1::u)
        else (f1,f2)::cl , u
     ) ([],[]) env.delta
  in
  List.fold_left assume {env with delta=cl} u
```

We note that all operations on Γ are efficient thanks to hash-consing since the comparison of two proxies is a constant time operation. The main function of the solver performs the case splitting:

```
let rec unsat f env =
  try
    let env = assume env f in
    match env.clauses with
        [] → raise Sat
      | (a,b)::l →
          unsat a {env with delta=l};
          unsat (Cnf.mk_not a)
              (assume {env with delta=l} b)
  with Unsat → ()
```

Finally, the function `is_sat : formula → bool` checks the satisfiability of a formula:

```
let is_sat f =
  try
    unsat (cnf f) {gamma=S.empty;delta=[]}; false
  with Sat → true
```

3.2.4 Benchmarks

We perform some quick benchmarks of this SAT solver on two different kinds of valid formulas whose sizes are parameters. The first one is due to de Bruijn and claims that, given an odd number of boolean variables on a circular list, there are at least two adjacent variables with the same value:

$$deb(n) = \left(\bigwedge_{i=0}^{2n}(p_i \leftrightarrow p_{i+1 \bmod 2n}) \rightarrow c\right) \rightarrow c$$

The second one is the famous pigeon-hole principle saying that if $n+1$ pigeons occupy n holes then at least one hole contains two pigeons:

$$ph(n) = \left(\bigwedge_{p=1}^{n+1}\bigvee_{h=1}^{n} x_{p,h}\right) \rightarrow \bigvee_{h=1}^{h}\bigvee_{p=1}^{n+1}\bigvee_{q=1}^{n+1} x_{p,h} \wedge x_{q,h}$$

Figures 2 and 3 show the timings for two versions of the SAT solver, one with hash-consing as presented in the previous sections and one with disabled hash-consing. As we can see, turning hash-consing on is always a winning strategy. On the first example, we even observe a different asymptotic behavior.

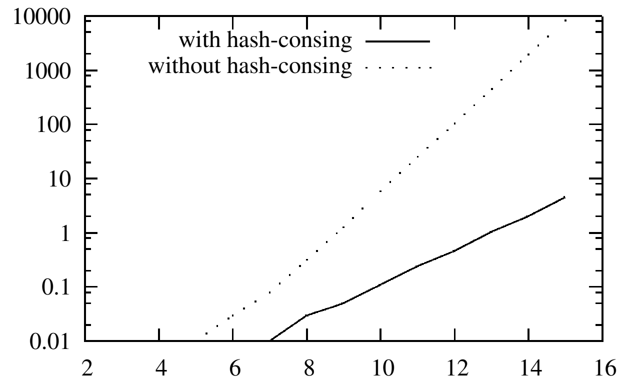

Figure 2. Benchmarking $deb(n)$ formulas

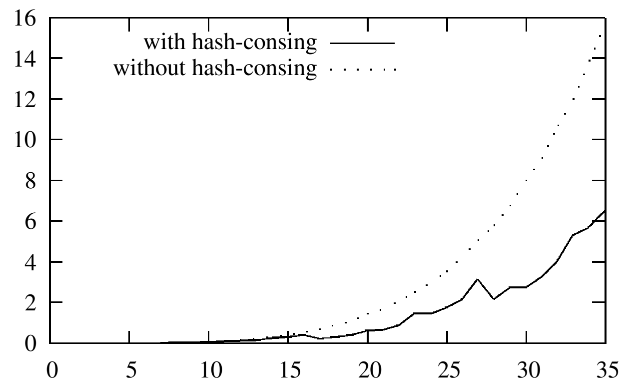

Figure 3. Benchmarking $ph(n)$ formulas

3.3 Binary Decision Diagrams

In this section, we show how to use our hash-consing library to quickly implement a Binary Decision Diagrams (BDD) package [5]. The purpose is not to build a competitive BDD library but to show how two main features of BDDs, namely sharing of equal trees and dynamic programming, are provided for free by our library. A BDD is a dag representing a boolean formula. It is either

- `Zero`, representing *false*,
- `One`, representing *true*, or
- `Node`(v, l, h), representing the formula $(v \wedge l) \vee (\neg v \wedge h)$, where v is a variable and l and h two BDDs.

A BDD is said to be *reduced* and *ordered* (with respect to a given order over variables) whenever the following properties hold:

- on any path from the root to a leaf, the variables respect the given order;
- no two distinct nodes are structurally equal (maximal sharing);
- no node `Node`(v, l, h) is such that $l = h$.

One fundamental property of (reduced ordered) BDDs is that each propositional formula has a unique representation. As a consequence, a formula represented as a BDD is valid if and only if it is reduced to `One` and is satisfiable if and only if it is not reduced to `Zero`. Said otherwise, all the computation is concentrated in the BDD building.

In the following, we show how our hash-consing library removes most of the implementation difficulty in the design of a simple BDD library (see for instance Andersen's lecture notes [3] for an introduction to BDD algorithms). To build a datatype of hash-consed values, we follow the same steps as in Section 2.2. We first define the OCAML type for BDDs:

```
type variable = int
type bdd =
  node hash_consed
and node =
  | Zero | One
```

```
apply op b1 b2 =
  if b1 and b2 are 0 or 1 then return op b1 b2
  if var b1 = var b2 then
    mk (var b1) (apply op (low b1) (low b2))
                (apply op (high b1) (high b2))
  else if var b1 < var b2 then
    mk (var b1) (apply op (low b1) b2)
                (apply op (high b1) b2)
  else
    mk (var b2) (apply op b1 (low b2))
                (apply op b1 (high b2))
```

Figure 4. Applying a boolean operation to two BDDs

```
  | Node of variable × bdd × bdd
```

Then we apply the `Hashcons.Make` functor to suitable equality and hashing functions to get an implementation of hash-consing tables:

```
module Node = struct
  type t = node
  let equal x y = ...
  let hash x = ...
end
module HC = Hashcons.Make(Node)
```

Then we introduce a global table to store the BDD nodes and the corresponding hash-consing function:

```
let table = HC.create 251
let hashcons = HC.hashcons table
```

Finally we define the smart constructors for BDDs:

```
let zero = hashcons htable Zero
let one = hashcons htable One
let mk v low high =
  if low == high then low
  else hashcons htable (Node (v, low, high))
```

This last constructor takes care of building reduced BDDs and maximal sharing is ensured by the hash-consing function.

To turn a propositional formula into a BDD, it is useful to rely on the following generic operation

```
val apply :
  (bool → bool → bool) → bdd → bdd → bdd
```

which applies an arbitrary binary boolean operator to two BDDs while maintaining the variables ordering invariant. Then we will have the usual connectives $\wedge$, $\vee$ and $\rightarrow$ as particular instantiations. The algorithm for `apply` is quite simple and is given Figure 4, where var, low and $high$ are the accessors for the three fields of `Node`. To get an efficient implementation of this algorithm, one has to add dynamic programming i.e. to remember the results of previous calls, as we did Section 3.1. With BDDs as hash-consed terms, it is immediate to build a hash table indexed by a pair of BDDs (using physical equality and a $O(1)$ hash function based on tags):

```
module H2 = Hashtbl.Make(
  struct
    type t = bdd × bdd
    let equal (u1,v1) (u2,v2) = ...
    let hash (u,v) = ...
  end)
```

and to use it to implement dynamic programming in `apply`:

```
let apply op b1 b2 =
  let cache = H2.create 251 in
  let rec apply u1 u2 =
    try
      H2.find cache (u1,u2)
    with Not_found →
      let res = ... in
      H2.add cache (u1,u2) res; res
  in
  apply b1 b2
```

Putting all together, we get a minimal BDD library fitting in less than 80 lines of OCAML code. If it cannot be compared with a state-of-the-art BDD library, its performances are not so bad, as shown on the de Bruijn ($deb(n)$) benchmark:

n	100	200	300	400	500
time in seconds	0.19	2.25	7.99	18.5	30.4

and on the pigeonhole ($ph(n)$) benchmark:

n	10	11	12	13	14
time in seconds	2.28	7.45	22.1	52.0	150

4. Conclusion

We have presented an OCAML implementation of a type-safe modular hash-consing library. Our approach has many practical benefits with respect to an ad hoc implementation. First, it introduces a type distinction between hash-consed values and normal values, that statically ensures maximal sharing. Then, this library is parameterized by a user equality that allows to identify terms according to an arbitrary equivalence relation. Finally, the library tags hash-consed values with unique integers that can be used to improve data structures such as hash tables, sets and dictionaries.

Hash-consing is also used in slightly different contexts such as ML runtimes where it is performed in a systematic way by the garbage collector [4, 10]. Our approach is less intrusive and the programmer is free to use hash-consing at relevant places in his code.

Our library is a free software available at `http://www.lri.fr/~filliatr/software.en.html`, together with implementations of sets and dictionaries using Patricia trees following [12]. It is already used in several OCAML applications (regular expression library, rewriting toolbox, first-order decision procedure). It would be interesting to see its effect on other existing OCAML applications such as the Coq proof assistant.

Our library should be easily translated to any other ML dialect. Indeed, `x == y` can be replaced by `x.tag = y.tag`, with only a small efficiency loss, when pointer equality is not available. Furthermore, when weak pointers are not available, traditional hash-tables can be substituted to weak hash-tables, at the extra cost of possible memory leaks.

There are still open issues. First, values always have to be constructed before being searched for in the hash-consing table, even when they happen to be already allocated. Thus the garbage collector still has to reclaim values that were unnecessarily allocated. To avoid this pitfall, we could have one hash-consing function for each constructor but such a library would require meta-programming.

Serialization is another issue. Indeed, the uniqueness of tags cannot be preserved when hash-consed values are written to files and reloaded afterwards. The user must implement his own input/output functions to rehash all serialized values (either by saving untagged values or by ignoring tags at loading time).

Last, the pattern-matching over hash-consed values is less convenient, especially for deep patterns. It could be easily solved with a native support for views [13] in OCAML.

Acknowledgments

We thank the anonymous referees for their detailed comments and suggestions.

References

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

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

[3] Henrik Reif Andersen. An Introduction to Binary Decision Diagrams. Lecture notes, 1998. `http://www.itu.dk/people/hra/`.

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

[5] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 8(C-35):677–691, 1986.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[8] A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.

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

[10] 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.

[11] Gérard Huet. Constructive Computation Theory. Courses notes. Available at `http://pauillac.inria.fr/~huet/CCT/`.

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

[13] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *14th ACM Symposium on Principles of Programming Languages*, Munich, January 1987.