Implementing hash-consed structures in Coq

Thomas Braibant, Jacques-Henri Jourdan, David Monniaux

Implementing hash-consed structures in Coq*

Thomas Braibant† Jacques-Henri Jourdan† David Monniaux‡

April 22, 2013

Abstract

We report on three different approaches to use hash-consing in programs certified with the Coq system, using binary decision diagrams (BDD) as running example. The use cases include execution inside Coq, or execution of the extracted OCaml code. There are different trade-offs between faithful use of pristine extracted code, and code that is fine-tuned to make use of OCaml programming constructs not available in Coq. We discuss the possible consequences in terms of performances and guarantees.

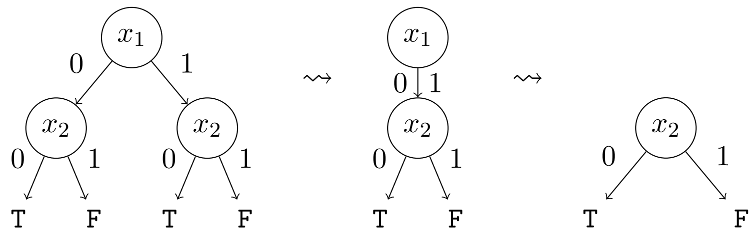
1 Introduction

Hash-consing is an implementation technique for immutable data structures that keeps a single copy, in a global hash table, of semantically equivalent objects, giving them unique identifiers and enabling constant time equality testing and efficient *memoization* (also known as *dynamic programming*). A prime example of the use of hash-consing is reduced ordered binary decision diagrams (ROBDDs, BDDs for short), representations of Boolean functions [3] often used in software and hardware formal verification tools, in particular *model checkers*.

A Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be represented as a complete binary tree with $2^n - 1$ decision nodes, labeled by variables x_i according to the depth from the root (thus the adjective *ordered*) and with subtrees labeled 0 and 1, and leaves labeled `T` (for true) or `F` (for false) . Such a tree can be *reduced* by merging identical subtrees, thus becoming a connected *directed acyclic graph* (see second diagram below); choice nodes with identical children are removed (see third diagram below). The reduced representation is *canonical*: a function is (up to variable ordering $x_1, \dots, x_n$) represented by a unique ROBDD.

For instance, the function $f(0,0) =$ `T`, $f(0,1) =$ `F`, $f(1,0) =$ `T`, $f(1,1) =$ `F` is represented, then simplified as:

*This work was partially funded by ANR project "VERASCO" and ERC project "STATOR"
†INRIA
‡CNRS / VERIMAG



In practice, one directly constructs the reduced tree. To do so, a BDD library usually maintains a global pool of diagrams and never recreates a diagram that is isomorphic to one already in memory, instead reusing the one already present. In typical implementations, this pool is a global hash table. Hence, the phrase *hash consing* denotes the technique of replacing nodes creation by lookup in a hash table returning a preexisting object, or creation of the object followed by insertion into the table if previously nonexistent. A unique identifier is given to each object, allowing fast hashing and comparisons. This makes it possible to do efficient *memoization*: the results of an operation are tabulated so as to be returned immediately when an identical sub-problem is encountered. For instance, in a BDD library, memoization is crucial to implement the or/and/xor operations with time complexity in $O(|a|.|b|)$ where $|a|$ and $|b|$ are the sizes of the inputs; in contrast, the naive approach yields exponential complexity.

In this article, we investigate how hash-consing and memoization, imperative techniques, may be implemented using the Coq proof assistant, using the example of a BDD library, with two possible uses: 1) to be executed inside Coq with reasonable efficiency, e.g. for proofs by reflection; 2) or to be executed efficiently when extracted to OCaml, e.g. for use in a model-checking or static analysis tool proved correct in Coq.

2 A Problem and Three Solutions

In the following, we propose to implement a BDD library using three different approaches. We focus on a minimal set of operations: node creation, Boolean operations (or, and, xor, not) and equality testing on formulas represented as ROBDDs; and we provide formal guaranties of their correctness. (Note that, in some of our solutions, we do not prove the completeness of the equality test. That is, we prove that the equality test returning true implies equality of the formulas; but proving the converse is not essential for many applications.)

The typical way of implementing hash-consing (a global hash table) does not translate easily to Coq. The reason is that the Gallina programming language at the heart of the Coq proof assistant is a purely applicative language, without imperative traits such as hash tables, or pointers or pointers equality.

Therefore, there are two approaches to the implementation of hash-consing for data-structures in Coq. The first one is to model the memory using finite maps inside Coq, and use indices in the maps as surrogates for pointers, implementing all the aforementioned operations on these persistent maps. Such an implementation was described in [5, 4], and we propose a new one in §2.1. The second one is to recover imperative features by fine-tuning the extraction of Coq code: either by realizing selected Coq constants by efficient OCaml definitions, e.g., extracting Coq

```
Inductive expr :=
  F | T | N : positive → expr.
Definition node := (expr * var * expr).
Record hashcons := {
graph: positive ⇝ node;
hmap : node ⇝ positive;
next : positive }.
```

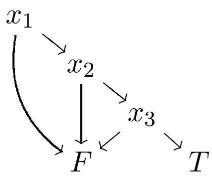
```
Definition mk_node (l : expr) (v: var) (h : expr) st :=
if expr_eqb l h then (l,st)
else match find (l,v,h) (hmap st) with
       | Some x ⇒ (N x, st)
       | None ⇒ (N st.(next), upd (l,v,h) st)
     end.
```

Figure 1: Hash-consing in pure Coq

constructors into smart OCaml constructors and fixpoint combinators into memoizing fixpoint combinators (see §2.2); or by explicitly declaring as axioms the OCaml code implementing the hash constructs and its properties (see §2.3).

2.1 Pure Coq

Our first implementation of BDDs is defined as follows in Coq. First, we assign a unique identifier to each decision node. Second, we represent the directed acyclic graph underlying a BDD as a Coq finite map from identifiers to decision nodes (that is, tuples that hold the left child, the node variable and the right child). For instance, the following graph, on the left, can be represented using the map on the right.



$$\begin{array}{lll} 1 & \mapsto & (\mathrm{F},\ x_1,\ \mathrm{N}\ 2) \\ 2 & \mapsto & (\mathrm{F},\ x_2,\ \mathrm{N}\ 3) \\ 3 & \mapsto & (\mathrm{F},\ x_3,\ \mathrm{T}) \end{array}$$

Then, we implement the hash-consing pool using another map from decision nodes to node identifiers and a `next` counter that is used to assign a unique identifier to a fresh node. Equality between BDDs is then provided by decidable equality over node identifiers. We present on Fig. 1 our inductive definitions (left) and the code of the associated allocation function `mk_node` (right), knowing that `upd n st` allocates the fresh node `n` in the hash-consing state `st` (taking care of updating both finite maps and incrementing the "next fresh" counter).

We define well-formedness as follows. A node identifier is *valid* in a given global state when it is lower than the value of the `next` counter. Then, the notion of well-formedness of global states covers the facts that `graph` maps all valid node identifiers to valid nodes (nodes whose children are valid); and `hmap` is a left-inverse of `graph`.

Then, all operations thread the current global state in a monadic fashion that is, of course, reminiscent of a state monad. The correctness of BDD operations corresponds to the facts that 1) the global state is used in a monotonic fashion (that is the structure of the resulting global state is a refinement of the input one and that the denotation of expressions is preserved); 2) the resulting global state is well-formed; 3) the denotation of the resulting BDD expression is correct. As can be expected from our data structure, BDD operations cannot be defined using structural recursion (there is no inductive structure on which to recurse). Using well-founded recursion is difficult here because the well-founded relation involves both parameters of the function and the global state. Proving it to be well-founded would involve merging non-trivial proofs of monotonicity within programs. In the end, we resorted to define partial functions that use a *fuel* argument to ensure termination.

Finally, it is possible to enrich our hash-consing structure with memoization tables in order to tabulate the results of BDD operations.

```
Record memo := {
mand : (positive * positive) ⇝ expr;
mor  : (positive * positive) ⇝ expr;
mxor : (positive * positive) ⇝ expr;
mneg : positive ⇝ expr}.
```

```
Record BDD := { ... :> hashcons; ... :> memo}.
```

The memoization tables are passed around by the state monad, just as the hash-consing structure. It is then necessary to maintain invariants on the memoization information. Namely, we have to prove that the nodes referenced in the domain and in the codomain of those tables are valid; and that the memoization information is semantically correct.

As a final note: this implementation currently lacks garbage collection (allocated nodes are never destroyed until the allocation map becomes unreachable as a whole); it could be added e.g. by reference counting.

2.2 Smart constructors

In the previous approach, we use a state monad to store information about hash-consing and memoization. However, one can see that, even if these programming constructs use a mutable state, they behave transparently with respect to the pure Coq definitions. If we abandon efficient executability inside Coq, we can write the BDD library in Coq as if manipulating decision trees without sharing, then add the hash-consing and memoization code by tweaking the extraction mechanism. An additional benefit is that, since we use native hash tables, we may as well use *weak* ones, enabling the native garbage collector to reclaim unused nodes without being prevented from doing so by the pointer from the table.

More precisely, we define our BDDs as in Fig. 2a. Moreover, we tell Coq to extract the `bdd` inductive type to a custom `bdd` OCaml type (see left of Fig. 2b) and to extract constructors into smart constructors maintaining the maximum sharing property. These smart constructors make use of generic hash-consing library by Conchon and Filliâtre [2] that defines the α `hash_consed` type of hash-consed values of type α and the `hashcons` function that returns a unique hash-consed representative for the parameter. Internally, the library uses suitable hash and equality functions on BDDs together with weak hash tables to keep track of unique representatives.

In Coq, we define the obvious `bdd_eqb` function of type `bdd` $\rightarrow$ `bdd` $\rightarrow$ `bool`, that decides structural equality of BDDs. Then, we extract this function into OCaml's physical equality. From a meta-level perspective, the two are equivalent thanks to the physical unicity of hash-consed structures.

The last ingredient needed to transform a decision tree library into a BDD library is memoization. We implement it by using special well-founded fixpoint combinators in Coq definitions, which we extract into a memoizing fixpoint combinator in OCaml. As an example, we give the definition of the `bdd_not` operation in Fig. 2c. The fixpoint combinator is defined using the Coq general `Fix` well-founded fixpoint combinator that respects a fixpoint equality property. The definition of `bdd_not` then uses `memoFix1` and requires proving that the BDDs sizes are decreasing (these trivial proof obligations are automatically discharged).

We extract the `memoFix1` combinator to a memoizing construct, that is observationally equivalent to the original one. However, this new construct tabulates results

```
Inductive bdd: Type :=
| T | F | N: var → bdd → bdd → bdd.
```

```
Extract Inductive bdd ⇒
  "bdd hash_consed" ["hT" "hF" "hN"] "bdd_match".
```

(a) BDDs in Coq as decision trees

```
type bdd =
| T | F
| N of var * bdd hash_consed * bdd hash_consed

let bdd_tbl = hashcons_create 257
```

```
let hT = hashcons bdd_tbl T
let hF = hashcons bdd_tbl F
let hN (p, b1, b2) = hashcons bdd_tbl (N(p, b1, b2))

let bdd_match fT fF fN b =
  match b.node with
  | T → fT () | F → fF ()
  | N(p, b1, b2) → fN p b1 b2
```

(b) Hash-consed OCaml BDD type

```
Definition memoFix1 :=
  Fix (well_founded_ltof bdd bdd_size).
Lemma memoFix1_eq : ∀ Ty F b,
  memoFix1 Ty F b =
  F b (fun b' _ ⇒ memoFix1 Ty F b').
Proof. [...]   Qed.
```

```
Program Definition bdd_not : bdd → bdd :=
  memoFix1 _ (fun b rec ⇒ match b with
                | T ⇒ F | F ⇒ T
                | N v bt bf ⇒
                  N v (rec bt _) (rec bf _)
              end).
```

(c) Using a fixpoint combinator for `bdd_not`

Figure 2: Implementing BDDs in Coq, extracting them using smart constructors

in order to avoid unnecessary recursive calls. We use similar techniques for binary operations. As all Coq definitions are kept simple, proofs are straightforward: we can prove semantic correctness of all operations directly using structural induction on decision trees.

2.3 Axioms

In the previous approach, hash-consing and memoization are done after the fact, and are completely transparent for the user. In the following, we make more explicit the hypotheses that we make on the representation of BDDs. That is, we make visible in the inductive type of BDDs that each BDD node has a "unique identifier" field (see Fig. 3) and we take the node construction function as an axiom, which is implemented in OCaml. Note that nothing prevents the Coq program from creating new BDD nodes without calling this function `mkN`. Yet, only objects created by it (or copies thereof) satisfy the `valid` predicate; we must declare another axiom stating that unique identifier equality is equivalent to Coq's Leibniz equality *for valid nodes*. Then, we can use unique identifiers to check for equality.

This approach is close to the previous one. It has one advantage, the fact that unique identifiers are accessible from the Coq code. They can for instance be used for building maps from BDDs to other data, as needed in order to print BDDs as a linear sequence of definitions with back-references to shared nodes. Yet, one could also expose unique identifiers in the "smart constructor" approach by stating as axioms that there exists an injection from the BDD nodes to a totally ordered type of unique identifiers.

The use of axioms is debatable. On the one hand, the use of axioms somewhat lowers the confidence we can give in the proofs, and they make the code not executable within Coq. On the other hand, these axioms are actually used implicitly when extracting Coq constructors to "smart constructors": they correspond to the

```
Axiom var : Set.

Axiom uid : Set.
Axiom uid_eqb : uid → uid → bool.
Axiom uid_eq_correct: ∀ x y : uid,
    (uid_eqb x y =true) ↔ x=y.

Inductive bdd : Set :=
| T | F
| N : uid → var → bdd → bdd → bdd.

Axiom mkN : var → bdd → bdd → bdd.

Axiom mkN_ok :
∀ v : var, ∀ bt bf : bdd,
  ∃ id, mkN v bt bf = N id v bt bf.
```

```
Inductive valid : bdd → Prop :=
| valid_T : valid T
| valid_F : valid F
| valid_N : ∀ var bt bf,
  (valid bt) → (valid bf) →
  (valid (mkN var bt bf)).

Axiom shallow_equal_ok :
  ∀ id1 id2 : uid,
  ∀ var1 var2 : var,
  ∀ bt1 bf1 bt2 bf2 : bdd,
    valid (N id1 var1 bt1 bf1) →
    valid (N id2 var2 bt2 bf2) →
    id1 = id2 →
    N id1 var1 bt1 bf1 =
    N id2 var2 bt2 bf2.
```

Figure 3: Axiomatization of equality using unique identifiers

metatheoretical statement that these constructors behave as native Coq constructors. Thus, they make explicit some of the magic done during extraction.

3 Discussion

We compare our approaches on different aspects:

Executability inside Coq. Both the "smart constructors" and the "pure" implementations can be executed inside Coq, even if the former has dreadful performances (when executed inside Coq, it uses binary decision trees). The "axiomatic" approach cannot be executed inside Coq.

Efficiency of the extracted OCaml code. We have yet to perform extensive testing, but preliminary benchmarks indicate that the "pure" approach yields code that is roughly five times slower than the "smart constructors" approach (and we assume that the latter is also representative of the "axiomatic" approach) on classic examples taken from previous BDD experiments in Coq [5]. We have yet to measure memory consumption.

Trust in the extracted code. Unsurprisingly, the "smart constructors" and the "axiomatic" approaches yield code that is harder to trust, while the "pure" approach leaves the extracted code pristine.

Proof. From a proof-effort perspective, the "smart constructors" is by far the simplest. The "axiomatic" approach involves the burden of dealing with axioms. However, it makes it easier to trust that what is formally proven corresponds to the real behavior of the underlying runtime. By comparison, the "pure" approach required considerably more proof-engineering in order to check the validity of invariants on the global state.

Garbage collection. Implementing (and proving correct) garbage collection for the "pure" approach would require a substantial amount of work. By contrast, the "smart" and "axioms" approaches make it possible to use OCaml's garbage collector to reclaim unreachable nodes.

4 Conclusion and directions for future works

In this paper, we proposed two solutions to implement hash-consing in programs certified with the Coq system. The first one is to implement it using Coq data-structures; the second is to use the imperative features provided by OCaml through the tuning of the extraction mechanism. The difference in flavor between the mapping of Coq constants to smart OCaml realizers or the axiomatization of there realizers in Coq is a matter of taste. In both cases, some meta-theoretical reasoning is required and requires to "sweep something under the rug".

We conclude with directions for future works. First, we believe that the smart constructors approach is generalizable to a huge variety of inductive types. One can imagine that it could be part of the job of Coq's extraction mechanism to implement on-demand such smart constructors and memoizers as it was the case for other imperative constructs [1]. Second, we look forward to investigate to what extent one could provide a certified version of the hash-consing library proposed by Conchon and Filliâtre [2].

Ackowledgements. We thank the reviewers for their helpful comments and Jean-Christophe Filliâtre for fruitful discussions.

References

[1] Michaël Armand et al. "Extending Coq with Imperative Features and Its Application to SAT Verification". In: *Proc. ITP*. Vol. 6172. LNCS. Springer, 2010, pp. 83–98. DOI: 10.1007/978-3-642-14052-5_8.

[2] Sylvain Conchon and Jean-Christophe Filliâtre. "Type-Safe Modular Hash-Consing". In: *ACM SIGPLAN Workshop on ML*. Portland, Oregon, Sept. 2006. URL: `http://www.lri.fr/~filliatr/ftp/publis/hash-consing2.pdf`.

[3] Donald E. Knuth. "The Art of Computer Programming". In: vol. 4A. Binary decision diagrams. Addison-Wesley, 2011. Chap. 7.1.4. ISBN: 978-0201038040.

[4] Kumar Neeraj Verma and Jean Goubault-Larrecq. *Reflecting BDDs in Coq*. Anglais. Rapport de recherche RR-3859. INRIA, 2000. HAL: inria-00072797.

[5] Kumar Neeraj Verma et al. "Reflecting BDDs in Coq". In: *Proc. ASIAN*. Vol. 1961. LNCS. Springer, 2000, pp. 162–181. ISBN: 3-540-41428-2.