



HAL
open science

Implementing and reasoning about hash-consed data structures in Coq

Thomas Braibant, Jacques-Henri Jourdan, David Monniaux

► **To cite this version:**

Thomas Braibant, Jacques-Henri Jourdan, David Monniaux. Implementing and reasoning about hash-consed data structures in Coq. 2013. hal-00881085v1

HAL Id: hal-00881085

<https://inria.hal.science/hal-00881085v1>

Preprint submitted on 7 Nov 2013 (v1), last revised 25 Sep 2015 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementing and reasoning about hash-consed data structures in Coq*

Thomas Braibant[†] Jacques-Henri Jourdan[†] David Monniaux[‡]

November 2013

Abstract

We report on four different approaches to implementing hash-consing in Coq programs. The use cases include execution inside Coq, or execution of the extracted OCaml code. We explore the 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. We use the running example of binary decision diagrams and then demonstrate the generality of our solutions by applying them to other examples of hash-consed data structures.

1 Introduction

Hash-consing is a programming technique used to share identical immutable values in memory, keeping a single copy of semantically equivalent objects. It is especially useful in order to get a compact representation of abstract syntax trees. A hash-consing library maintains a global pool of expressions and never recreates an expression equal 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 node *construction* by a lookup in a hash table returning a preexisting object, or creation of the object followed by insertion into the table if previously nonexistent. This makes it possible to get *maximal sharing* between objects, if hash-consing is used systematically when creating objects.

Moreover, a unique identifier is given to each object, allowing fast hashing and comparisons. This makes it possible to do efficient *memoization* (also known as *dynamic programming*): the results of an operation are tabulated so as to be returned immediately when an identical sub-problem is encountered.

*This work was partially funded by ANR project “VERASCO” and ERC project “STATO R”

[†]INRIA

[‡]CNRS / VERIMAG

Hash-consing and memoization are examples of imperative techniques that are of prime importance for performance. The typical way of implementing hash-consing (a global hash table) does not translate easily into 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, pointers, or pointer equality. In the following, we discuss how hash-consing can be implemented using the Coq proof assistant, with two possible use cases in mind:

- execution inside Coq with reasonable efficiency, e.g. for proofs by reflection;
- efficient execution when extracted to OCaml, e.g. for use in a model-checking or static analysis tool proved correct in Coq.

More precisely, we present a few *design patterns* to implement hash-consing and memoization on various examples, and we use binary decision diagrams (BDDs) as our prime example.

Outline. This paper is organized as follows. Since hash-consing can be seen as a particular kind of memoization, where one memoizes constructor applications, we begin this paper with a short review of the techniques that are available in Coq in order to memoize functions (§2). Then, we give a short primer on BDDs in §3 and present the “menu” of our four implementations of BDD libraries in §4. These implementations are subsequently described in §5 and §6 and use different design patterns. We discuss the pros and cons of each design pattern in §7. We adapt these design patterns to implement strong reduction of hash-consed lambda-terms in §8. We discuss related work and conclude this paper in §9.

2 Memoization in Coq: state of the art

Generally speaking, there are two ways to use Coq code: either to execute it inside Coq, e.g., to define reflective decision procedures; or to extract it to OCaml (or other supported languages), then compile the extracted code. These two usages impose constraints on the possible design patterns. In the remainder of this section, we describe what the state of the art in memoization approaches is with respect to these two use cases.

2.1 State monad

The first idea to implement memoization in Coq is to use a dedicated data structure to tabulate function calls: that is, use a finite map of some sort to store the pairs (*argument*, *result*) for a given function. Users of a library

```

Section t.
  Context {A : Type}.
  CoInductive lazy : Type :=
  | Think : A → lazy.

  CoInductive lazies : Type :=
  | Cons : lazy → lazies → lazies.

  Fixpoint nth n cst :=
  match n, cst with
  | 0, Cons (Think xi) _ ⇒ xi
  | S p, Cons _ c ⇒ nth p c
  end.

  CoFixpoint mk_lazy {B} f (n : B) :=
  Think (f n).

  CoFixpoint memo' f n :=
  Cons (mk_lazy f n) (memo' f (S n)).

  Definition memo f := memo' f 0.
End t.

(* A costly version of the identity *)
Fixpoint big {A} n : A → A :=
match n with
| 0 ⇒ fun x ⇒ x
| S n ⇒ fun x ⇒ big n (big n x)
end.

Definition id n := big n n.
Definition k := memo id.
Definition run x := nth x k.

```

Figure 1: Memoization using co-inductives

implemented in this fashion must, then, thread this state through the program using a state monad.

While this solution always works, it has non-negligible verification cost: we must prove that the bindings in the memoization table are correct, i.e., the values held in the table correspond to the result of the would-be corresponding function calls.

Then, if we memoize a function of type $A \rightarrow B$, we get a function of type $A \rightarrow M B$, where M is the type constructor associated with our state monad. Even if the latter seems equivalent to the former, these types are different: therefore, this tiny modification (memoizing a function) requires modifying every caller of the memoized function.

Since it is cumbersome to perform all computations inside a state monad, we continue this section with a review of other (partial) solutions to the memoization problem.

2.2 Shallow memoization

It is possible in some cases to make a *shallow embedding* of this memoization table in Coq. Melquiond [14] describes how to use co-inductive objects as a cache that stores previously computed values of a given function. In this section, we present his clever idea in Fig. 1, with a few cosmetic changes¹. A `lazy` value is defined as a co-inductive, which will effectively be represented as a `think` when executed by Coq’s virtual machine, and will be extracted to

¹We found out that similar definitions actually made their way into Coq’s standard library under the name `StreamMemo.v`, yet we believe that this idea deserves more explanation than what is available.

OCaml as a `Lazy.t` thunk. A *thunk* is a term whose evaluation is frozen until it is actually needed (lazy evaluation), then the computed value is cached in the thunk so that it is instantly returned in case the term is evaluated again (thus, performing a kind of memoization).

Then, `lazies` represents a stream (an infinite list) of thunks, that can be peeked at using `nth`. Each time the virtual machine has to destruct the `cst` argument of `nth`, it will remember the two arguments of this cons cell. Then, it suffices to build a stream `memo` that computes the value of a function `f` for each possible natural number: in practice, `f` will be evaluated lazily, and only once per input value.

As an example, the following snippet of code memoizes the time-consuming function `id`. (Remark that it also demonstrates that the intermediate calls of `id` are not memoized.)

```
Time Eval vm_compute in run 26. (* 6s      *)
*)
Time Eval vm_compute in run 26. (* 0s      *)
*)
Time Eval vm_compute in run 25. (* 3s      *)
*)
Time Eval vm_compute in run 25. (* 0s      *)

Time Eval vm_compute in id 25. (* 3s *)
Time Eval vm_compute in id 25. (* 3s *)
```

Note that this clever trick is not limited to functions over Peano numbers, and could be adapted for branching structures, e.g. memoizing functions over binary numbers. Unfortunately, it seems hard to adapt this idea to memoize recursive calls: consider for instance the function

```
Fixpoint exp n := match n with 0 => 1 | S n => exp n + exp n end.
```

In order to compute `exp` in linear time² we need a memoizing fixpoint combinator rather than a way to memoize a predefined function.

(To be complete, let us add that functions `f : nat → A` that can be expressed as the iteration of a given function `g : A → A`, can actually be memoized using the aforementioned technique. While we obviously could rewrite `exp` to fit into that scheme, this is not the case of the other functions that we use through our developments.)

This problem is somewhat representative of the one that we have to face when it comes to BDDs: first, we need memoizing fixpoint combinators; second, the function that we memoize depends on the global state of the hash-consing machinery, which evolves through times. In both cases, we fall out of the scope of the above trick: there is no predefined function to memoize.

²Notwithstanding the fact that we are using Peano numbers here.

2.3 Memoization in extracted code

Now, we turn to a partial solution to the memoization problem, that works when the user is solely interested in executing the code extracted from a Coq program.

Vafeiadis introduced recently a restricted form of mutable state called *adjustable references* [17]. The idea is that adjustable references store some internal value that can only be updated in innocuous ways. That is, adjustable references are equipped with an observation function; and the update function ensures that the result of the observations remain equals through updates. It is therefore possible to adjust the values held in the references in a way that changes, e.g., the costs of some computations, yet does not change the result of these computations.

Vafeiadis demonstrated how to use adjustable references to memoize a function f by tabulating its results: the adjustable reference internal state is a finite map, while the observation function simply returns f . Updating the contents of the finite map does not change the observation function; but subsequent reads may make use of the fact that a value was previously computed and stored inside the memoization cache. We refer the reader to Vafeiadis' article for more details about this idea.

Remark that it is again the case that this technique does not scale to the definition of memoizing fixpoint combinators. Therefore, adjustable references do not quite fit our needs either.

2.4 Summary

Out of the three memoization techniques available in Coq that we described, only the second one suits our need: it makes it possible to define memoizing fixpoint combinators. In this paper, we use this *state monad* approach, then we will come up with a new idiom.

3 A primer on binary decision diagrams

In the following, we use reduced ordered binary decision diagrams (ROBDDs, BDDs for short) as a running example of hash-consed data structures. BDDs are representations of Boolean functions and are often used in software and hardware formal verification tools, in particular *model checkers* [13].

The data structure. A Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ can be represented as a complete binary tree with $2^n - 1$ decision nodes, labeled by variables x_i according to the depth i from the root (thus the adjective *ordered*); edges are labeled 0 and 1; leaves are labeled T (for true) or F (for false). The semantics of such a diagram is: $f(x_1, \dots, x_n)$ is obtained by

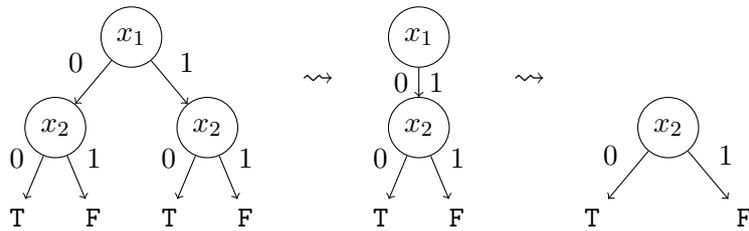


Figure 2: The BDD for the function $f(0, 0) = \text{T}$, $f(0, 1) = \text{F}$, $f(1, 0) = \text{T}$, $f(1, 1) = \text{F}$

traversing from the root and following the 0 or 1 edge of a node labeled by x_i according to the value of x_i .

Such a tree can be *reduced* by merging identical subtrees, thus becoming a connected *directed acyclic graph* (DAG; see second diagram in Fig. 2); furthermore, decision nodes with identical children are removed (see third diagram in Fig. 2). These transformations preserve semantics. The reduced representation is *canonical*: given a variable ordering x_1, \dots, x_n , a function is represented by a unique ROBDD.

Building BDDs. In practice, one directly constructs the reduced tree; let us see how. All BDD nodes allocated so far are stored in a global hash table, so that a new object is created only if not already in the hash table: *there are never two different, yet identical, objects in memory at the same time*, a crucial invariant that must be maintained throughout the execution.

Each BDD node is given a unique identifier: for instance, the current value of a global 64-bit counter incremented following each allocation³. Since a new object is never created if an identical object already exists, two objects are equal if and only if they have the same unique identifier. The equality test, which is usually expensive over tree structures, since it requires full traversal, is instead implemented by a very fast comparison of unique identifiers.

Unique identifiers are also used for hashing. When attempting to construct a node (v, b_0, b_1) where v is the variable, b_0 is the subtree labeled 0 and b_1 is the subtree labeled 1, one computes the hash value of the node as $H(v, u_0, u_1)$ where u_0 and u_1 are the respective unique identifiers of the subtrees b_0 and b_1 . This hash value is then used to look up the node in the hash table. Again, such shallow hashing is considerably faster than having to traverse the tree structures.

³In certain implementations, the unique identifier is the address of the node; this supposes that objects never change address, which is not the case in OCaml.

Operations on BDDs. The principal way to build BDDs is to combine the diagrams of two functions f and g in order to obtain the BDD for other functions such as $f \wedge g$, $f \vee g$ or $f \oplus g$. The basic combinator that underlies binary operations on BDDs is called *melding* [13].

Suppose that we have two nodes $\alpha = (v, l, h)$ and $\alpha' = (v', l', h')$. The melding of α and α' , denoted by $\alpha \diamond \alpha'$ is defined as follows:

$$\alpha \diamond \alpha' = \begin{cases} (v, l \diamond l', h \diamond h') & \text{if } v = v' \\ (v, l \diamond \alpha', h \diamond \alpha') & \text{if } v < v' \\ (v', \alpha \diamond l', \alpha \diamond h') & \text{if } v > v' \end{cases}$$

Then, the binary operation \diamond is entirely defined by the results of

$$F \diamond \alpha \quad \alpha \diamond F \quad T \diamond \alpha \quad \alpha \diamond T$$

For instance, the conjunction operation $f \wedge g$ can be defined by melding the BDDs for f and g , and using the following rewrite rules for the leaf cases:

$$F \diamond \alpha \rightarrow F \quad \alpha \diamond F \rightarrow F \quad T \diamond \alpha \rightarrow \alpha \quad \alpha \diamond T \rightarrow \alpha$$

For the sake of clarity, we focus on binary operations in the following: they are representative of the difficulties we had to face⁴.

Memoization. The fact that each node has a unique identifier also makes it possible to memoize the results of BDD operations. One keeps a map from sub-problems (a pair of nodes α and α') to nodes (the result of $\alpha \diamond \alpha'$), so as to be returned immediately when a previously solved sub-problem is encountered.

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: as the function is executed, its results on the subtrees of the original problem are stored in a structure indexed by (u_a, u_b) where u_a and u_b are the unique identifiers of the a and b inputs. In contrast, the naive approach has exponential complexity, since the function may be evaluated exponentially often on the same couple of subtrees.

4 Implementing BDDs in Coq.

The following two sections §5 and §6 describe a total of *four* implementations of BDDs in Coq. To make things clear for the reader, we give each of these implementation a reference name, a pointer to the relevant section of the paper and a short description.

⁴Note that we only need two rewrite rules for commutative binary operations.

PURE-DEEP. See §5.1. A pure Coq implementation of BDDs that makes a deep embedding of memory as finite maps and uses indices as surrogates of pointers.

PURE-SHALLOW. See §5.2. A pure Coq implementation of BDDs that uses a shallow embedding of memory.

SMART. See §6.1. An "impure" implementation of BDDs in Coq: we implement hash-consing and memoization through the extraction mechanism of Coq.

SMART+UID. See §6.2. A variation on the previous approach in which we discuss how to expose and axiomatize the operations on the unique identifiers associated with BDD nodes.

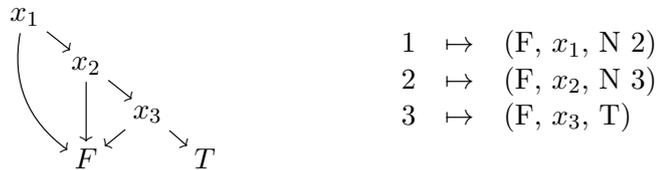
5 Pure solutions

As the reader can gather from the summary above, we describe two pure Coq implementations of BDDs libraries in this section, that differ mainly by the way we model the memory and the allocation of nodes.

5.1 The PURE-DEEP approach

The idea here is to model the memory using finite maps inside Coq and use indices in the maps as surrogates for pointers, implementing BDD operations as manipulations of these persistent maps. Such an implementation was described in [19, 18], but we propose a new one here for the sake of completeness, and because the old one did not age well w.r.t. the evolution of Coq. Our implementation is defined as follows.

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.



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. In the situation above, `next` is equal to 4 and the hash-consing map is defined as follows

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

Notations. Through this paper, we stay as close as possible to our Coq code. Yet we allow us some liberty when it comes to finite map (that are pervasives in our code): the notation $\mathfrak{t} \rightsquigarrow \mathfrak{s}$ denotes a type of efficient maps from type \mathfrak{t} to type \mathfrak{s} ; and we use indiscriminately the notations `get` and `set` that respectively access and update such finite maps. Implicitly, these two notations have the following types:

$$\begin{aligned}
\text{get} &: \mathbb{A} \rightarrow (\mathbb{A} \rightsquigarrow \mathbb{B}) \rightarrow \text{option } \mathbb{B} \\
\text{set} &: \mathbb{A} \rightarrow \mathbb{B} \rightarrow (\mathbb{A} \rightsquigarrow \mathbb{B}) \rightarrow (\mathbb{A} \rightsquigarrow \mathbb{B})
\end{aligned}$$

(Note that we do use the efficient finite maps from the Coq standard library; but we chose here to abstract from the particular module definitions and names that we have to use in our code, for the sake of legibility.)

Fig. 3 shows our inductive definitions and the code of the associated allocation function `mk_node`, 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). Then, equality between BDDs (`eqb`) is provided by decidable equality over node identifiers.

All operations thread the current global state in a monadic fashion. 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, see Fig. 4); 2) the resulting global state is well-formed; 3) the denotation of the resulting BDD expression is correct.

Invariants and well-formedness properties. We present the well-formedness properties we preserve over expressions (`wf_expr`) and over hash-consing states (`wf_hashcons`) in in Fig. 5. The inductive predicate `wf_expr st v e` depends on a variable level `v` that indicates a bound over the variable identifiers appearing in the BDD `e`. This ensures that variable are kept ordered. Then, a well-formed expression is either a leaf (cases `wf_T` and `wf_F`) or a pointer to a hash-consing node with an head-variable `w` less than the bound `v`. (Remark that this definition of well-formed expression is not directly recursive, because the actual recursion takes place in the definition of `wf_st` below; this notion of well-formed expressions coupled with well-formedness of the state suffice to prove that expressions are DAGs with a suitable ordering on the variables.)

The predicate `wf_hashcons` is the general well-formedness property over hash-consing states. The predicate `wf_bijection` ensures that the two maps `hmap` and `graph` form a bijection; `wf_expr_lt_next` ensures that the next

```

Inductive expr := F | T | N : positive → expr.

Definition eqb a b :=
  match a,b with | T,T ⇒ true | F,F ⇒ true | N x, N y ⇒ (x ==? y) | _, _ ⇒ false end.

Definition node := (expr * var * expr).

Record hashcons := {
  graph : positive ~> node;
  hmap  : node ~> positive;
  next  : positive
}.

Definition upd u st := {
  graph := set (next st) u (tmap st);
  hmap  := set u (next st) (hmap st);
  next  := (next st + 1)
}.

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

```

Figure 3: Hash-consing in pure Coq

```

Record incr (st1 st2 : hashcons) : Prop := {
  incr_lt : (next st1 ≤ next st2);
  incr_find : ∀ p x, get p (graph st1) = Some x → get p (graph st2) = Some x
}.

```

Figure 4: Monotonicity predicate over hash-consing states

fresh variable counter will indeed produce fresh variables; `wf_map_wf_expr_l` and `wf_map_wf_expr_r` ensure that a node stored in the hash-consing structure has well-formed children, respecting the variable order; `wf_reduced` ensures that all those nodes are reduced (ie. their left child is different from their right child).

Note that the statement that the hash-consing structure is correct corresponds to `wf_bijection` and `wf_expr_lt_next`. The other statements correspond to properties about BDDs, namely the facts that they are reduced and ordered.

The denotation of BDD expressions. Recall that BDD expression, as we have defined them in this section, do not have an inductive structure

```

Inductive wf_expr st : var → expr → Prop :=
| wfe_T : ∀ v, wf_expr st v T
| wfe_F : ∀ v, wf_expr st v F
| wfe_N : ∀ (p : positive) (l : expr) (w : var) (h : expr),
    get p (graph st) = Some (l, w, h) →
    ∀ (v : var), (w < v) →
    wf_expr st v (N p).

Record wf_hashcons (st : hashcons) : Prop := {
  wf_bijection : ∀ p n, get n (hmap st) = Some p ↔ get p (graph st) = Some n;
  wf_expr_lt_next : ∀ p v, wf_expr st v (N p) → (p < next st);
  wf_map_wf_expr_l : ∀ p x v y, get p (graph st) = Some (x, v, y) → wf_expr st v x;
  wf_map_wf_expr_h : ∀ p x v y, get p (graph st) = Some (x, v, y) → wf_expr st v y;
  wf_reduced : ∀ p l v h, get p (graph st) = Some (l, v, h) → l <> h
}.

```

Figure 5: Well-formedness of hash-consing state

```

Inductive value env st : expr → bool → Prop :=
| value_T : value env st T true
| value_F : value env st F false
| value_N : ∀ (p : positive) (l : expr) (v : var) (h : expr),
    get p (graph st) = Some (l, v, h) →
    ∀ (vv : bool), get env v = Some vv →
    ∀ (vhl : bool), value env st (if vv then h else l) vhl →
    value env st (N p) vhl.

```

Figure 6: The denotation of BDD expressions

we can recurse on: this means that we cannot define the semantics of BDD expressions as a Coq fixpoint. Rather, we have to define the semantics of BDD expressions as an inductive predicate that uses the state of the hash-consing data structure to go through the graph. The inductive `value` is shown on Fig. 6. It is defined as a binary relation with two parameters (the valuation of variables `env` and the state of the hash-consing structure `st`) and two arguments: the expression and its denotation (a Boolean).

Termination of BDD operations. The first problem that we have to solve in our Coq representation is that, 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). Unfortunately, we cannot easily use well-founded recursion here because the well-founded relation involves both parameters of the function and the global state.

The problem is that the termination of the BDD operations relies on the fact that the graph of nodes is acyclic; but the graph is not fixed through an execution of the melding operation! Rather, the global state is threaded in the

```

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

Record state := { ... :> hashcons; ... :> memo}.

```

Figure 7: Adding memoization tables to the global state

recursive calls of \diamond . Therefore, proving the recursive calls to be well-founded requires to prove that the \diamond operation is monotonic w.r.t. graph inclusion. In order to prove termination, we would have to prove to define the fixpoint and prove this monotonicity property *at the same time*. This would involve embedding invariants directly in the global state, using dependent types. This has two major drawbacks: first, defining terms containing complex dependent types is cumbersome and proving properties about them is often very challenging. Second, we would have to pay close attention to prevent Coq from normalizing those proof terms if we want to use this library for reflection purposes.

In the end, we resorted to define partial functions that use a *fuel* argument to ensure termination: that is, they use an explicit bound on the number of iterations to do.

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

The memoization tables (see Fig. 7) are passed around by the state monad, just as the hash-consing structure. It is then necessary to maintain invariants on the memoization information.

We present the invariants over the memoization maps for the binary operations⁵ in Fig. 8. First, we have to prove that the nodes referenced in the domain and in the codomain of those tables are well-formed and that those tables keep the bounds over the variables correct. Then, we have to state that the memoization information is semantically correct. One downside of this data structure definition is that we are forced to define one table per operation that we want to memoize, and that this is not modular: adding a new operation requires modifying the definition of the memoization state and add the corresponding field in the `wf_memo` record.

(Note that finding the correct pattern of memoization for a program is still an art rather than a science: using the data structure above, we keep the memoized values from one run of an operation to the other. In this section, we will settle on this conservative strategy, but other memoization strategies are possible that yield different performance and memory consumption profiles.)

⁵The case of the negation operation is similar and is not detailed.

```

Record wf_memo2 (st : hashcons) (m : (positive * positive) ~> expr)
  (opb : bool → bool → bool)
:= {
  wf_memo2_find_wf_res :
    ∀ x y e, get (x, y) m = Some e → wf_expr st v (N x) → wf_expr st v (N y) →
      wf_expr st v e;
  wf_memo2_find_wf :
    ∀ x y e, get (x, y) m = Some e → ∃ v, wf_expr st v (N x) ∧ wf_expr st v (N y);
  wf_memo2_find_sem :
    ∀ x y res, get (x, y) m = Some res →
      ∀ env vx vy, value env st (N x) vx → value env st (N y) vy →
        value env st res (opb va vb)
}.

Record wf_memo_neg (st : hashcons) (m : positive ~> expr) := { ... }.

Record wf_memo (st : state) := {
  wf_memo_mand : wf_memo2 st (mand st) Datatypes.andb;
  wf_memo_mor : wf_memo2 st (mor st) Datatypes.orb;
  wf_memo_mxor : wf_memo2 st (mxor st) Datatypes.xorb;
  wf_memo_mneg : wf_memo_neg st (mneg st)
}.

Record wf_st (st : state) : Prop := { ... : wf_hashcons st; ... : wf_memo b }.

```

Figure 8: Invariant over the memoization information

A mouthful of code. The final version of our code is shown on Fig. 9. We use a do-notation à la Haskell to make it more palatable.

Then, we prove that under some hypotheses, this combinator is correct: that is, it produces well-formed hash-consing structures and memoization tables, and the denotation of the resulting expression correspond is correct. For the sake of clarity, we will not expose these hypotheses nor the resulting correctness theorem in this paper, and refer the interested reader to the archive of code that accompanies this paper [5].

Implementing the and operation. However, we demonstrate the use of this combinator on the particular example of the **and** function; all other binary operations follow the same pattern. First, we have to define a function `upd_and` that updates the memoization state: it is simply a wrapper that add an element to the right memoization table, and leave the others untouched. Then, we define the function `mk_and` as a simple call to the binary combinator. The crux here is the choice of the functions `Fx` and `Tx` that specify the behavior of the combinator at the leaves of the DAG.

```

Definition upd_and na nb res (st : state) := Definition mk_and :=
  mk_state st
  { | mand := set (na,nb) res (mand st);
    mor := mor st;
    mxor := mxor st;
    mneg := mneg st
  } |}.
  combinator
  (fun x st => Some (F,st) ) (* Fx *)
  (fun x st => Some (x,st) ) (* Tx *)
  (fun a b st => get (a,b) (mand st))
  upd_and.

```

Then, the semantic correctness theorem for this operation is defined as follows.

```

Theorem mk_and_sem_correct depth env (st : state) a b va vb :
  wf_st st → value env st a va → value env st b vb →
  ∀ res st', mk_and depth a b st = Some (res, st') →
    value env st' res (andb va vb).

```

Canonicity. We can prove that this representation of BDDs is canonical: that is, well-formed equivalent expressions are mapped to the same nodes.

```

Definition equiv st e1 e2 :=
  ∀ env v1 v2, value env st e1 v1 → value env st e2 v2 → v1 = v2.

```

```

Lemma canonicity st v e1 e2 :
  wf_st st → wf_expr st v e1 → wf_expr st v e2 →
  equiv st e1 e2 → e1 = e2.

```

From this result, it follows that the (non-recursive) `eqb` function from Fig. 3 is a correct and complete characterization of semantic equivalence of expressions.

```

Lemma eqb_correct st v e1 e2 :
  wf_st st → wf_expr st v e1 → wf_expr st v e2 →
  (eqb e1 e2 = true ↔ equiv st e1 e2).

```

Section combinator.

```
Variable fx : expr → state → option (expr * state).  
Variable tx : expr → state → option (expr * state).  
Variable memo_get : positive → positive → state → option expr.  
Variable memo_update : positive → positive → expr → state → state.
```

```
Definition memo_node a b l v h st :=  
  let (res, st) := mk_node l v h st in  
  let st := memo_update a b res st in  
  (res,st).
```

Fixpoint combinator depth a b st :=

```
match depth with  
| 0 ⇒ None  
| S depth ⇒  
  match a,b with  
  | F, _ ⇒ fx b st  
  | _, F ⇒ fx a st  
  | T, _ ⇒ tx b st  
  | _, T ⇒ tx a st  
  | N na, N nb ⇒  
    match memo_get na nb st with  
    | Some p ⇒ Some (p,st)  
    | None ⇒  
      do nodea ← get na (graph st);  
      do nodeb ← get nb (graph st);  
      let '(l1,v1,h1) := nodea in  
      let '(l2,v2,h2) := nodeb in  
      match Pos.compare v1 v2 with  
      | Eq ⇒  
        do x, st ← combinator depth l1 l2 st;  
        do y, st ← combinator depth h1 h2 st;  
        Some (memo_node na nb x v1 y st)  
      | Gt ⇒  
        do x, st ← combinator depth l1 b st;  
        do y, st ← combinator depth h1 b st;  
        Some (memo_node na nb x v1 y st)  
      | Lt ⇒  
        do x, st ← combinator depth a l2 st;  
        do y, st ← combinator depth a h2 st;  
        Some (memo_node na nb x v2 y st)  
      end  
    end  
  end  
end.  
end.
```

End combinator.

Figure 9: Combinator for binary operations

Garbage collection. In the above version of BDDs, we have not implemented garbage collection. That is, allocated nodes are never destroyed, until the allocation map becomes unreachable as a whole. Garbage collection could be added, e.g., using a stop and copy operation that preserve a set of roots. This is beyond the scope of this paper.

5.2 The PURE-SHALLOW approach

The previous implementation uses a *deep embedding* of the representation of the BDD in memory via the `graph` map. This is a natural way to encode a directed acyclic graph, but, as we saw, makes it difficult (if not unfeasible) to deal properly with termination. Therefore, we would like to be able to reason about BDDs as if they formed an inductive type, while keeping the ability to share sub-terms at runtime.

There is actually no need to look further than inductive types to do that. The standard intuition about inductive types is that they define the smallest type closed under application of constructors: the mental image that we get is from that is a *tree*. Yet, there is nothing that prevent us to use the system to share sub-terms.⁶

In this section, we demonstrate that we can encode BDDs in Coq using a representation that looks like binary decision *trees*, yet has runtime performances similar to the PURE-DEEP implementation (see §5.1), using explicit sharing. We present on Fig. 10 our inductive definitions, and the associated allocation function `mk_node`.

The code of the `hc_node` function is subtle: a call to `hc_node e bdd` will perform a lookup in the hash-consing map `hmap`: if the same expression was previously allocated, then we return the old version; otherwise, we update the map `hmap` with a mapping from the expression to its hash-consed version. The lookup ensures that equal expressions (modulo the comparison function used to index `hmap`) are mapped to the same hash-consed expression. Then, `mk_node l v h bdd` will first test the identifiers of `l` and `h` for equality: if it is the case, then there is no need to introduce a new node; otherwise, we perform a call to `hc_node`.

As an example, assume that `x` and `y` are expressions with different identifiers. The following code

```
let (a, bdd) := mk_node x v y bdd in
let (b, bdd) := mk_node x v y bdd in
let (c, bdd) := mk_node a v' b bdd in ...
```

⁶Such an approach may be interesting even if each structure is a tree, but different trees can share sub-trees. For instance, the Astrée static analyzer implements maps as balanced binary trees with *opportunistic* sharing [3, §6.2]: e.g. when an idempotent operation $f(x, x) = x$ is to be performed on the images of two maps g and h , returning the map $x \mapsto f(g(x), h(x))$, then if g and h are determined to be identical through pointer equality, g is directly returned. Thus, applying such a function to two maps that differ only on few values is considerably faster than traversing the two maps.

```

Inductive expr := | F | T | N : hc_expr → var → hc_expr → expr
with hc_expr := HC : expr → positive → hc_expr.

(* Two extra definitions that are used as coercions in the following code*)
Definition unhc (e : hc_expr) := let 'HC res _ := e in res.
Definition ident (e : hc_expr) := let 'HC _ res := e in res.

Definition eqb a b :=
  match a,b with
  | T,T | F,F ⇒ true
  | N (HC _ id1) x (HC _ id2), N (HC _ id1') x' (HC _ id2') ⇒
    (id1 =? id1') && (x =? x') && (id2 =? id2')
  | _, _ ⇒ false
  end.

Record hashcons := {hmap : expr ↪ hc_expr; next : positive}.

Definition upd u st :=
  let r := HC u (next b) in
  (r, { | hmap := set u r (hmap b); next := (next b + 1) |}).

Definition hc_node (e : expr) (bdd : BDD) :=
  match get e (hmap bdd) with
  | Some x ⇒ (x, bdd)
  | None ⇒ upd e bdd
  end.

Definition mk_node (l : hc_expr) (v : var) (h : hc_expr) bdd :=
  if (ident l =? ident h) then (l,bdd) else hc_node (N l v h) bdd.

```

Figure 10: A shallow-embedding of sharing

will make `a`, `b` and `c` point to the same memory location! However, if `x` and `y` where not shared maximally, then neither are `a`, `b` nor `c`.

A word on memoization. There is no difference at all in the way we handle memoization in this implementation w.r.t. §5.1. That is, we implement the same `state` record as before; and pass the same memoization tables around.

A word on termination. It is now easier to define recursive functions that operate on BDDs by taking advantage of the inductive definition of `expr`. We have to stress that *easier* is not *easy* because the Coq termination checker requires that recursive calls are made on a structurally smaller argument: there is no builtin support for recursive definitions with pairs of arguments that are decreasing w.r.t. a lexicographic order. Therefore, we have to use nested fixpoints or prove that the recursive calls are well-founded. In this section, we choose the former.

Re-implementing the combinator. We are now ready to describe the code of the implementation of the binary combinator presented in Fig. 11. The code is similar to the one in Fig. 9 with a few key differences: thanks to the inductive definition of expressions, we do not have to perform lookups in the `graph` map and we do not have to use fuel anymore. This makes the combinator function *total*; and we can get rid of the Maybe monad.

Canonicity. Again, we prove that this representation of BDDs is canonical: well-formed equivalent expressions are mapped to the same nodes. Again, we have the corollary that the (non-recursive) equality test from Fig. 10 that inspects the (top-level) node identifiers is a complete and correct characterization of semantic equivalence.

Definition `equiv e1 e2 :=`
 $\forall \text{env } v1 \ v2, \text{value env } e1 \ v1 \rightarrow \text{value env } e2 \ v2 \rightarrow v1 = v2.$

Lemma `eqb_correct st e1 e2 v :`
 $\text{wf_st } st \rightarrow \text{wf_expr } st \ v \ e1 \rightarrow \text{wf_expr } st \ v \ e2 \rightarrow$
 $(\text{eqb } e1 \ e2 = \text{true} \leftrightarrow \text{equiv } e1 \ e2).$

Comparison with the previous approach. The implementation presented in this section is derived from the previous one, with the following improvements: the proofs are roughly 20% shorter; the performances are slightly better when executing the code inside Coq (there is less administrative book-keeping to do in our data structures); the functions that operates on BDDs are total. Furthermore, it would probably be easier to implement garbage collection in this setting than in the previous one, thanks to the simpler definition of the global state.

```

Section combinator.
Variable fx : hc_expr → BDD → hc_expr * BDD.
Variable tx : hc_expr → BDD → hc_expr * BDD.
Variable memo_get : positive → positive → BDD → option (hc_expr).
Variable memo_update : positive → positive → hc_expr → BDD → BDD.

Fixpoint combinator (a : hc_expr) :=
  fix combinator_rec (b : hc_expr) bdd :=
  match memo_get (ident a) (ident b) bdd with
  | Some p ⇒ (p,bdd)
  | None ⇒
    let '(res, bdd) :=
      match unhc a, unhc b with
      | F, _ ⇒ fx b bdd
      | _, F ⇒ fx a bdd
      | T, _ ⇒ tx b bdd
      | _, T ⇒ tx a bdd
      | N l1 v1 h1, N l2 v2 h2 ⇒
        match Pos.compare v1 v2 with
        | Eq ⇒
          let '(x, bdd) := combinator l1 l2 bdd in
          let '(y, bdd) := combinator h1 h2 bdd in
          mk_node x v1 y bdd
        | Gt ⇒
          let '(x, bdd) := combinator l1 b bdd in
          let '(y, bdd) := combinator h1 b bdd in
          mk_node x v1 y bdd
        | Lt ⇒
          let '(x, bdd) := combinator_rec l2 bdd in
          let '(y, bdd) := combinator_rec h2 bdd in
          mk_node x v2 y bdd
        end
      end
    in
    let bdd := memo_update (ident a) (ident b) res bdd in
    (res, bdd)
  end.
End combinator.

```

Figure 11: Node combinator, using shallow sharing

The situation is borderline ideal for the equality test: we prove that the (non-recursive) equality function that inspects the toplevel identifiers of nodes is a correct and complete characterization of semantics equivalence of BDD expressions. However, we have no way to prove that it corresponds to physical equality. Actually, we cannot *state* that it is never the case that two identical representations of the same term coexists, even if we could argue at a meta-level that it is indeed not the case.

6 From pure data structures to persistent data structures via extraction

In the previous section, 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.

We have seen earlier (see §2.3) that if we abandon (efficient) executability inside Coq, we can express new idioms. In the following, we implement the BDD library in Coq as if manipulating decision trees with neither sharing, nor hash-consing tables, nor memoization tables, then add the hash-consing and memoization code by tweaking the extraction mechanism.

6.1 The smart approach

More precisely, we define our BDDs as binary decision trees (see Fig. 12a), and implement operations in Coq on this simple data structure. Then, we tell Coq to extract the `bdd` inductive type to a custom `bdd` OCaml type (see left of Fig. 12b) and to extract constructors into *smart constructors* that maintain the maximal sharing property. The type defined in OCaml is identical to the Coq one, except that it carries one extra field of type `tag`, morally containing the associated unique identifier. The smart constructors make use of the hash-consing library used in Why3 [16], a recent version of a library by Conchon and Filliâtre [7]. It defines the `Hashcons.Make` functor, that we instantiate. The generated module provides a `HCbdd.hashcons` function that returns a unique hash-consed representative for the parameter.

The reader may notice that we choose to name `bdd` in Coq what is clearly a representation of a binary decision tree, and which corresponds to what was previously named `expr`. We believe that this particular choice of name makes sense if we consider values of type `bdd` to represent Boolean functions.

Discussion: the status of the equality test. In Coq, we define the obvious recursive function `eqb` of type `bdd → bdd → bool`, that decides structural equality of BDDs. Then, we extract this function into OCaml’s physical equality.

```

Inductive bdd: Type :=
| T | F | N : var → bdd → bdd → bdd.
Extract Inductive bdd ⇒
"bdd" ["hT" "hF" "hN"] "bdd_match".

```

(a) BDDs in Coq as decision trees

```

type bdd =
| T of tag | F of tag | N of positive * bdd * bdd * tag

module HCbdd = Hashcons.Make(...)

let hT = HCbdd.hashcons (T Weaktbl.dummy_tag)
let hF = HCbdd.hashcons (F Weaktbl.dummy_tag)
let hN (v, t, f) = HCbdd.hashcons (N (v, t, f, Weaktbl.dummy_tag))

let bdd_match fT fF fN t =
match t with
| T _ → fT ()
| F _ → fF ()
| N (v, t, f, _) → fN v t f

```

(b) Hash-consed OCaml BDD type

Figure 12: Implementing BDDs in Coq, extracting them using smart constructors

From a meta-level perspective, we argue that the two are equivalent thanks to the physical unicity of hash-consed structures, provided that all values are constructed using our smart-constructors, which is the case if we create all nodes from code extracted from Coq (of course, handwritten OCaml code may break this invariant). The key point here is that the way we build terms enforces the fact that the BDDs they build are maximally shared.

Keeping the BDD reduced. One could be tempted to put in the smart constructor `hN` a piece of code that would enforce that BDDs are reduced. That is, it would not build a node if its two children were identical. This idea is wrong, because if such a node construction was avoided, a subsequent case analysis would not behave as predicted by the Coq, and this would lead to some inconsistencies between the Coq code and the OCaml extracted code. Instead, we have to use the following helper function, written in Coq:

```

Definition N_check (v : var) (bt bf : bdd) : bdd :=
  if eqb bt bf then bt else N v bt bf.

```

This function builds a node only when necessary, but it becomes impossible to prove that a case analysis on the return value of this function would reveal `bt` and `bf` as children.

Implementing the combinator. The last ingredient needed to transform a decision tree library into a BDD library is memoization. We use the same kind of ideas: we define our functions as if not memoized, but we use a special well-founded fixpoint combinator, that we extract to a memoizing fixpoint combinator. The details can be seen on Fig. 13: we declare an abstract type class `memoizer` of types `A` such that we know how to memoize

```

Parameter memoizer : ∀ A : Type, Type.
Existing Class memoizer.
Extract Constant memoizer "'key" ⇒ "'key Helpers_common.memoizer".

Definition memo A {H : memoizer A} P := @id (∀ x : A, P x).
Extract Inlined Constant memo ⇒ "Helpers_common.memo".
Arguments memo [A] {H} [P] _ _ . (* Set implicit arguments for memo*)

Definition memo_rec A {H : memoizer A} := @Fix A.
Extract Inlined Constant memo_rec ⇒ "Helpers_common.memo_rec".
Arguments memo_rec [A] {H} [R] Rwf [P] F x. (* Set implicit arguments for
memo_rec *)

```

(a) Coq part

```

type `key memoizer =
  { memo : α. int → ((`key → α) → (`key → α)) → (`key → α) }

let memo m f = m.memo 5 (fun _ → f)

let memo_rec m f = m.memo 5 (fun frec x → f x (fun y _ → frec y))

```

(b) OCaml part

Figure 13: Memoizing combinators

functions of type $\forall x : A, P x$. This is extracted in OCaml to the type of polymorphic fixpoint combinators, with an extra technical `int` parameter used to specify the initial size of the used hash map. In Coq, we then define a memoizing combinator `memo` and a memoizing fixpoint combinator `memo_rec` as if they were not using memoization, but we ask the extraction mechanism to map them to special OCaml functions, that make use of the type class instance given as parameter. These functions are observationally equivalent to the Coq ones, provided that the type class instance is correct, and that the memoized function is pure.

It is worth noting that directly exposing the type ``key memoizer` in Coq would be unsound, because this allows to use its instances in a non-terminating manner: the `memo_rec` wrapper makes sure the recursive calls are well-founded. Moreover, it is important to understand that we have not axiomatized these combinators. Instead, we give real implementations, semantically equivalent to their OCaml counterparts. This is important, because it then becomes clear that we do not introduce any logical inconsistencies in Coq, and these terms keep a computational content which could be very useful in proofs.

This part of the code is modular, and can be used to memoize functions of any domain. It is up to the user to give instances of the `memoizer` type

```

Parameter memoizer_N : memoizer N.
Existing Instance memoizer_N.
Extract Inlined Constant memoizer_N ⇒ "Helpers_common.memoizer_N".

```

(a) Coq part

```

module NHT =
  Hashtbl.Make (struct
    type t = coq_N
    let equal = (=)
    let hash = N.to_int
  end)
let memoizer_N =
  { memo = fun n f →
    let h = NHT.create n in
    let rec aux x =
      try NHT.find h x with Not_found →
        let r = f aux x in NHT.replace h x r; r
    in aux }

```

(b) OCaml part

Figure 14: An instance of `memoizer` for `N`

class: to do so, he can provide dedicated code, as shown in Fig. 14 for the type `N` of binary natural integers, using simple hashtables. Alternatively, for the `bdd` type, our typeclass instance is just an OCaml wrapper around the hash-consing library builtin memoization mechanisms.

Again, we use the example of the `bdd_and` operation, shown in Fig. 15. As in Fig. 11, we define this function using two nested fixpoints, in order to handle the special recursion scheme of this function. The definition of `bdd_and` uses `memo_rec` twice, in a nested fashion. This is needed in order to handle the special recursion scheme of this function (decreasing on one of its two parameters).

Garbage collection. The strategy we use in §5.2 for garbage collection is very naive: we kept everything alive, forbidding any garbage collection. Here, the hash-consing library we use allow the garbage collector to reclaim any node that is not referenced by the program, by using suitable weak hash tables. Moreover, it reclaims any memoized value that is associated with a dead key (we will not give more details here and refer the reader to [7]). While it avoids memory leaks, this strategy does not necessarily give the best performances (see §7). We believe the changes necessary to implement a new strategy are small, and do not involve rewriting the proofs: only OCaml code is involved.

It is important to note that we have to memoize the functions each argument after the other (keeping them curried). Indeed, one should be tempted to memoize a function `bdd * bdd → bdd`. This is not a good idea: in this case, the pairs containing the arguments are no longer accessible after the function call, so that the garbage collector is going to collect most of the

```

Definition bdd_and : bdd → bdd → bdd.
refine
  (memo_rec (well_founded_ltof _ bdd_size) (fun x ⇒
    match x with
    | T ⇒ fun _ y ⇒ y
    | F ⇒ fun _ _ ⇒ F
    | N xv xt xf ⇒ fun recx ⇒
      memo_rec (well_founded_ltof _ bdd_size) (fun y ⇒
        match y with
        | T ⇒ fun _ ⇒ x
        | F ⇒ fun _ ⇒ F
        | N yv yt yf ⇒ fun recy ⇒
          match Pos.compare xv yv with
          | Eq ⇒ N_check xv (recx xt _ yt) (recx xf _ yf)
          | Lt ⇒ N_check yv (recy yt _) (recy yf _)
          | Gt ⇒ N_check xv (recx xt _ y) (recx xf _ y)
          end
        end)
      end));
  unfold ltof; simpl; clear; abstract omega.
Defined.

```

Figure 15: The and operation on bdds

memoized data at each cycle. Using our pattern, we make sure a memoized datum is not reclaimed as long as both parameters are still alive in memory.

Discussion: comparison with previous approaches. In this instance of our BDD library, all Coq definitions are kept simple and proofs are straightforward. That is, we can prove semantic correctness of all operations directly using structural induction on decision trees and there is no state holding structures. There is a nice separation between the hash-consing and memoization code, that is generic, written in OCaml and not proved and the BDD code, mostly written and proved in Coq. We do not use monads in the Coq code, so that the interface stays modular and easy to use. Moreover, it is straightforward to implement garbage collection strategies in order to avoid memory leaks.

6.2 The smart+uid approach

The previous SMART approach totally hides the unique identifiers from the Coq code. Yet, exposing these unique identifiers may be useful at times.

Consider the following use case: from a BDD B we would like to build an equivalent propositional logic formula of *linear size*, for instance for feeding into a satisfiability modulo theory solver. In order to avoid an exponential blow-up, each shared sub-tree should generate one single sub-formula, used in a “let” binder so that its value can be used in multiple occurrences. The

obvious way to implement such a transformation is to first detect which sub-trees are shared, using a set of shared subtrees seen so far, then to perform the transformation, using a table of mappings from subtrees to bound variables.

It seems therefore highly desirable to be able to build sets and maps over our hash-consed type. Generic functional sets and maps are usually implemented using balanced trees over a totally ordered datatype; for efficiency, the comparison function should be very fast. An obvious choice would be to expose the unique identifiers to the Coq code (through a function $\text{bdd} \rightarrow \text{uid}$), or at least the total order that they induce (through a function $\text{bdd} \rightarrow \text{bdd} \rightarrow \text{comparison}$).

Unfortunately, doing so without precautions can lead to unsoundness. Consider a program where, in succession, two nodes A and B are allocated, then a node A' isomorphic to A is created; let $u_A < u_B$, $u_{A'}$ be the successive unique identifiers. If A is collected between the allocations of B and A' , then A' will be allocated, with $u_{A'} > u_B$. Yet, A' and A are, from the point of view of the Coq code, identical; thus $u_{A'} = u_A$, yielding an inconsistency.

The workaround is to use a normal hash-table, as opposed to a weak hash-table, which prevents the collection of unreachable nodes. Then, two identical nodes created within the same execution are necessarily physically equal and thus share the same identifier.

One difficulty remains. Gallina is a purely functional language; the evaluation of a given term always yields the same result, and one expects the same property to extend to the extracted OCaml program, as long as it does not interact with the external world (e.g. reading from files). Yet, this is not necessarily the case if one exposes the unique identifiers. Consider a program P_1 , such that the extracted OCaml code allocates two nodes A and B in this order. If P_1 is run standalone, then $u_A < u_B$. Yet, if another program P_2 allocating B' isomorphic to B is first run, and B' is not collected, then B' is the same as B and $u_B = u_{B'} < u_A$.

It seems questionable that the result of an evaluation should depend on whether or not some other (unrelated) evaluation has taken place, if only because it makes debugging difficult.⁷

⁷One could argue that, with certified programs in the Coq fashion, there is no need to debug: each function or module comes with a proof of its correctness, which compositionally entail the correctness of the whole program. Yet, commonly one only proves the results to be *correct*, not necessarily *optimal*, and one proves very seldom that the computation has the expected complexity. Furthermore, some computations are split between an untrusted solving procedure, and a trusted checker; a failed check entails having to debug the untrusted procedure, which may be hard if behaviors are hard to reproduce independently of the rest of program.

6.2.1 Maps and sets over hash-consed types

Arguably, the final result of an algorithm using functional sets and tables should not depend on the order relation used. One could thus opt to expose neither the unique identifiers, nor the order relation they induce, but only a functional map interface (with *find*, *remove* and *update*). Then, the results cannot depend on garbage collection and other disturbances to the ordering. Yet, one could also want these sets and maps to provide a *fold* operation; this would need to be restricted to associative and commutative operations (so that their result do not depend on the ordering). Such an interface would fit the *map-reduce* approach to aggregate objects.

One additional advantage is that, with this approach, it is possible to use weak hash tables and garbage collection, as long as the sets and maps point to the nodes used as keys and thus prevent their collection: the ordering relation is then exposed to the Coq code only through equality testing, be it explicit or implicit (through the use of set or maps).⁸

While this line of work seems powerful, we did not pursue it: the many axioms needed do not inspire confidence; some nontrivial meta-theoretic reasoning about the sequential execution of OCaml code extracted from Coq would be needed.

6.2.2 Axioms

In the SMART approach, hash-consing and memoization are done after the fact, and are completely transparent for the user; yet we felt a need to break this transparency by exposing the unique identifiers. In the following, we instead 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. 16) 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 SMART approach. It has one advantage, the fact that unique identifiers are directly accessible from the Coq code. The previously mentioned limitation applies: since unique identifiers are visible, one cannot use weak hash tables and the OCaml garbage collector.

⁸It is of paramount importance that the sets and maps use the actual nodes as keys (ordered by the order on unique identifiers), not just the unique identifiers: this prevents scenarios such as creating an object A , inserting a key u_A into some set S , collecting A , creating another object A' isomorphic to A , but with a different key $u_{A'}$, which could distinguished from A by membership in S . In a nutshell, it is crucial that the Coq program cannot test for equality of unique identifiers separated from their associated objects.

```

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

7 Discussion

In this section, we compare our design patterns on various aspects. Let’s start with the easy ones:

Executability inside Coq. The PURE-DEEP and PURE-SHALLOW implementations can be executed inside Coq, and have decent performances. The SMART approach can also be executed inside Coq, but has dreadful performances (when executed inside Coq, it uses binary decision trees). The SMART+UID approach cannot be executed inside Coq.

Trust in the extracted code. Unsurprisingly, the SMART and the SMART+UID approaches yield code that is harder to trust, while the PURE-DEEP and PURE-SHALLOW approaches leave the extracted code pristine.

Proof. From a proof-effort perspective, the SMART approach is by far the simplest. The SMART+UID 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-DEEP and PURE-SHALLOW approaches required

considerably more proof-engineering in order to check the validity of invariants on the global state. Note however that our proof arguments are much simpler in the latter one.

Garbage collection. Implementing (and proving correct) garbage collection for the PURE-DEEP or PURE-SHALLOW approaches would require a substantial amount of work. By contrast, the SMART approach make it possible to use OCaml’s garbage collector to reclaim unreachable nodes “for free”.

Operations As we have shown, binary operations can be handled with a single parametric combinator. All our implementations use such a combinator to implement conjunction, disjunction, exclusive-or and so on. There is little work to do to add support for any binary operation we may have overlooked. The situation is more complicated when it comes to ternary operations (such as the if-then-else Boolean operation). Implementing it using the SMART approach, and proving it correct requires around 80 lines of code. It would require a non-trivial amount of work to implement it using our PURE-DEEP or PURE-SHALLOW approaches, and we have not conducted this experiment yet. Other operations that are relevant in a BDD library are function composition and quantification. We have yet to implement these in any of our libraries.

7.1 Performances of the extracted code

We evaluate the performances of the OCaml code that is extracted from our “pure” (see §5) and “impure” (see §6) libraries, and we pit them against a reference library implemented in OCaml (available from Filliâtre’s web page). This reference library does not keep the memoization table alive from one execution of an operation to another: for instance, each time the `and` function is called, a new memoization hash-table is allocated. Therefore, to make up for a fair benchmark with our implementations, we modified this reference library to use a memoization strategy closer to ours. This alternative reference implementation is designed “reference (conservative)” in what follows.

Then, this evaluation, we use two standard benchmarks (see [19]). The first one is Urquhart’s formula $U(n)$ defined by

$$U(n) \triangleq x_1 \iff (x_2 \iff \dots (x_n \iff (x_1 \iff \dots (x_{n-1} \iff x_n))))$$

The second kind of formula states the pigeonhole principle $P(n)$ for $n + 1$ pigeons: that is, if there is $n + 1$ pigeons in n pigeonholes, then there is at least one hole that contains two pigeons. The plots of the execution time required to check that these formulas are tautology are given in Fig. 17.

First, we remark that the reference implementation that we use is (almost always⁹) faster than our best implementation SMART, by a factor that increases from 2 to 10.

Then, our PURE-DEEP implementation is roughly 4 times slower than the SMART one: interestingly enough this value is consistent over our benchmark, while we could have expected logarithmic factors to show up. Indeed, recall that the pure implementation uses functional finite maps, while the smart one uses OCaml hash-maps.

Also, we have run the same benchmark on our PURE-SHALLOW implementation, and the resulting plot almost matches the PURE-DEEP plot: the difference of execution times are consistently within a few percent of the PURE-DEEP execution time. We do not delve too much on this result: we wonder for instance to what extent changing the implementation of finite maps that we use could result in similar differences of performances.

Finally, remark that the reference implementation with the conservative memoization strategy is only twice faster than the SMART one on the pigeon-hole formulas. It is simply the case here that the particular memoization strategy one uses has an important impact on performances. We have not investigated to what extent we could modify the memoization strategy we use in our implementations to improve the execution time on these benchmarks. Note also that the performances of the two reference implementations are almost the same on the Urquhart benchmark:

In these benchmarks, we have not measured memory consumption, but we have capped memory usage to 6 GB. Data points that are missing on the plot for Urquhart's formula are problems that could not be solved with this memory bound. Memory usage is likely to be the limiting factor on bigger instances of these tests.

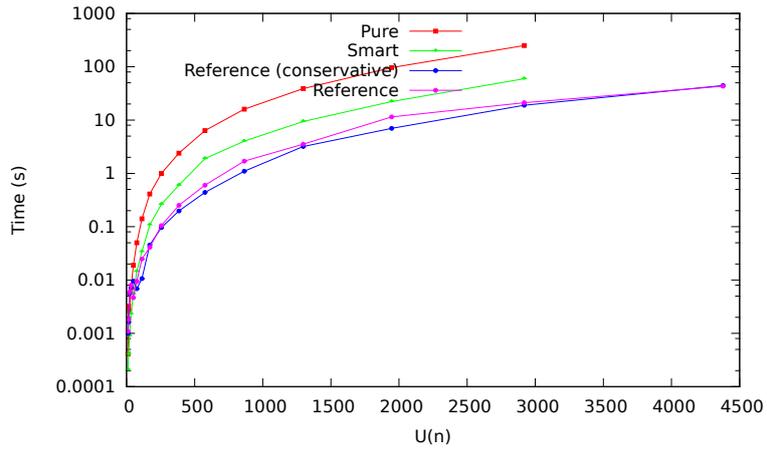
7.2 Maps and sets over hash-consed types

We argued in §6.2 that it may be necessary to implement efficient maps and sets over BDD nodes. In this section, we discuss how such maps and sets could be implemented in our various approaches.

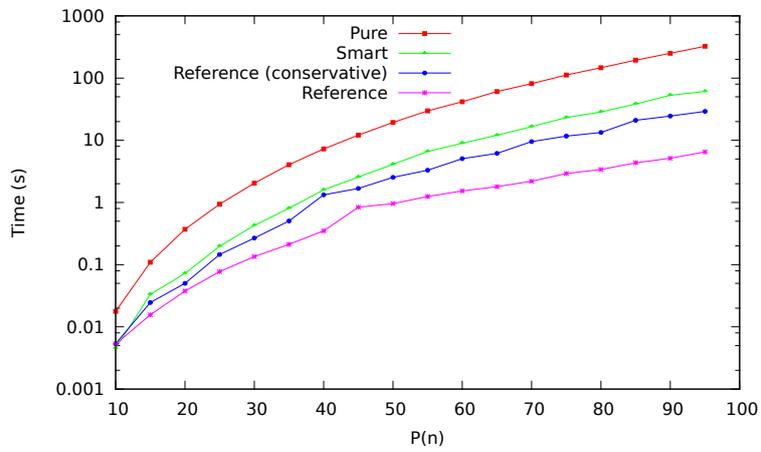
First, recall that our PURE-DEEP and PURE-SHALLOW expose the unique identifiers of the nodes to the user. Since these are plain Coq **positive** numbers, we can use Coq's efficient maps and sets over **positive**.

Second, the SMART+UID approach was designed specifically to expose unique identifiers and the suitable order relation in the Coq interface as an abstract type, at the expense of being unable to use weak hash-tables. This makes it possible to use Coq's efficient finite sets and finite maps. Alternatively, we argued that one can refrain from exposing the comparison function, and expose the signature of maps and sets.

⁹This is not necessarily true for small formulas due to differences in the startup costs of these libraries.



(a) Urquhart's formula



(b) Pigeonhole principle

Figure 17: Execution time for the BDD benchmarks

Finally, let’s consider the SMART approach. We can actually implement a comparison function on BDDs exactly as we implemented the operations on BDDs. That is, implement the corresponding “semantic” comparison function on BDTs, and extract it using our memoizing fixpoint combinators. This would make it possible to use Coq’s efficient finite sets and finite maps.

8 Implementing reduction in the λ -calculus

In the previous section, we have presented several design patterns about how to implement hash-consed data structures in Coq. In this section, we apply two of them to the example of reduction of λ -terms (the running example of Conchon and Filliâtre [7]).

8.1 An implementation without hash-consing

We first describe a Coq implementation, without memoization nor hash-consing, of λ -terms with de Bruijn indices inspired from Huet’s *Constructive Computation Theory* [12]. We implement four functions on terms: `lift`, `subst`, `hnf` (that puts a term in head-normal form) and `nf` (that puts a term in normal form). This amounts to roughly 60 lines of Coq code: we present the code for `lift` in Fig. 18b as a reference for future comparisons.

Then, we reuse Conchon and Filliâtre benchmark [7]: we implement a λ -calculus version of quicksort that operates on lists (encoded as λ -terms) of Church numerals; and apply this algorithm to sort the list $L = (0 :: 3 :: 5 :: 2 :: 4 :: 1 :: nil)$.

Using Coq’s virtual machine, sorting this list requires over 250 s on a recent desktop computer¹⁰: this involves a number of reduction that is exponential w.r.t. the size of the list and the size of the numbers. Extracting the code to OCaml makes it more efficient by a constant factor. We compile the extracted code with the native ocaml compiler, and run it to sort the list L .

time (s)	Number of allocated bytes	Max. size of the major heap (words)
30	$47 * 10^9$	$1.1 * 10^6$

The number of allocated bytes is given as reported by `Gc.allocated_bytes`: it corresponds to what is allocated during the execution of the sorting algorithm. The maximum size of the major heap is given as reported by OCaml `Gc.stat` function. In short, these numbers mean that the program allocates an awful lot of short-lived values.

¹⁰Equipped with Intel Xeon cores running at 3.60GHz.

```

Inductive term : Type :=
| Var : N → term
| App : term → term → term
| Abs : term → term.

```

(a) λ -terms with de Bruijn indices

```

Fixpoint lifti (n : N) (t : term) (k : N) :=
  match t with
  | Var i ⇒ if N.ltb i k then Var i else Var (i + n)
  | Abs t ⇒ Abs ( lifti n t (N.succ k) )
  | App t u ⇒ App (lifti n t k) (lifti n u k)
  end.

```

```

Definition lift n t := lifti n t 0.

```

(b) A implementation of `lift` without hash-consing nor memoization

```

Program Definition lifti : N → term → N → term :=
  memo (fun n ⇒
    memo_rec (well_founded_ltof _ size) (fun t rec ⇒
      memo (fun k ⇒
        match t with
        | Var i ⇒ if N.ltb i k then Var i else Var (N.add i n)
        | Abs t ⇒ Abs (rec t _ (N.succ k) )
        | App t u ⇒ App (rec t _ k) (rec u _ k)
        end))).
  (* Proof obligations elided *)

```

```

Definition lift n t := lifti n t 0.

```

(c) Implementing `lift` the smart way

Figure 18: Implementation of terms and `lift`

8.2 Using smart constructors

The SMART approach from §6.1 involves a small amount of changes to the code, mainly to use memoizing constructs and to add `memoizer` definitions. To get a rough idea of the changes involved, we present the modified code for `lift` in Fig. 18c: it is merely a matter of replacing abstractions and fixpoint definitions with their memoizing counterparts. The overall size of the code is now roughly over 80 lines of Coq code. While they do not make it possible to compute efficiently inside Coq, these changes have dramatic consequences on the execution time of the extracted code. Again, we compile the extracted code with the native OCaml compiler, and run it to sort the list L .

time (s)	Number of allocated bytes	Max. size of the major heap (words)
0.15	$0.12 * 10^9$	$3.1 * 10^6$

Here, we have a higher top size of the major heap, which account for our use of memoization tables. This is to be weighed against the lower amount of

allocated words and the 200× decrease in running time.

8.3 Using the shallow-embedding approach

Finally, we use the PURE-SHALLOW from §5.2. There is now roughly 300 lines of code and definitions to implement `lift`, `subst`, `hnf` and `nf`. In this section, we will make a brief review of this code, highlighting the differences with respect to what we presented in §5.2.

We give the modified definition of terms in Fig. 19a: each constructor takes an extra `positive` argument: that is, we drop the extra indirection we used in the definition of BDD expressions in §5.2. (This choice of definition yields simpler proof arguments because we do not need to perform mutual inductions on `expr` and `hc_expr`.)

We adapt the notion of global state from §5.2. Again, the `hmap` finite map is indexed using a non-recursive comparison function: it simply compare the head symbols of the terms and, if equal, compare the tags of the sub-terms. Again, we define one memoization table per function we wish to memoize.

Then, let us take the example of abstraction to describe term creation. We define the following smart constructor.

Definition `mk_Abs t st := mk_term (Abs t 1) st.`

The function `mk_Abs` is a simple wrapper around `mk_term`: it applies the term constructor `Abs` to the term `t` and a dummy unique identifier. Then, `mk_term` (see right of Fig. 19b) performs a lookup in the hash-consing table. If there exists a mapping from `Abs t p` (for some `p`) to `x` in the table, then we return `x`. Otherwise, the `upd` function replaces the dummy unique identifier with the next fresh unique identifier available, and it updates the map `hmap` with a mapping from the expression `t` to its tagged version `r`.

In effect, this code maintains the invariant that all the terms in the `hmap` table are the canonical representative of their equivalence classes modulo the term comparison used to index `hmap`. That is, we ensure that the terms we build verify the following property.

Definition `wf_term st t := get t (hmap st) = Some t.`

Let us jump to the definition of `lifti` (see Fig. 19c): it is defined using a well-founded fixpoint combinator `Fixm` (elided here) that ensures that the measure `msr` is decreasing through recursive calls. The bulk of this definition is `lifti_rec`: it is quite similar to the previous definitions of `lift`, except that we have to add fancy return clauses to the match definitions. They are necessary to be able to prove that the measure actually decreases through recursive calls. (Also, note that we could have inlined `lifti_rec` in `lifti`, but this two-step definitions makes it for more palatable proof goals later on).

The upside of these definition is that we are now able to compute the normal forms of our λ -terms inside Coq with decent efficiency: it takes

```

Inductive term : Type :=
| Var : N → positive → term
| App : term → term → positive →
      term
| Abs : term → positive → term.

Definition tag t n :=
match t with
| Var x _ ⇒ Var x n
| App t u _ ⇒ App t u n
| Abs t _ ⇒ Abs t n
end.

```

(a) Definition of terms, and the associated tagging function

```

Definition upd t (st : state) :=
let r := tag t (next st) in
(r, mk_state
  { |
    hmap := set t r (hmap st);
    next := (next st + 1)
  } (to_memo st)).

Definition mk_term (t : term) st :=
match get t (hmap st) with
| Some x ⇒ (x, st)
| None ⇒ upd t st
end.

```

(b) Updating the global state, and the term constructor

```

Let A := (term * N * state).
Let B := (term * state).
Let msr := fun (n : A) ⇒ size (fst (fst n)). (* measure function *)

Program Definition lifti_rec (n : N) (arg : A) (rec : ∀ x : A, msr x < msr arg → B)
:=
match arg as arg' return arg = arg' → B with
| (t,k,st) ⇒ fun Harg ⇒
  match get (n,ident t,k) (memo_lifti st) with
  | Some t ⇒ (t,st)
  | None ⇒ let (r,st) :=
    match t as t' return t = t' → _ with
    | Var i _ ⇒ fun H ⇒ if N.ltb i k
      then mk_Var i st
      else mk_Var (N.add i n) st
    | Abs t _ ⇒ fun H ⇒ let (t,st) := rec (t,(N.succ k),st) _ in
      mk_Abs t st
    | App t u _ ⇒ fun H ⇒ let (t,st) := rec (t,k,st) _ in
      let (u,st) := rec (u,k,st) _ in
      mk_App t u st
    end eq_refl
  in (r,upd_lifti (n,ident t,k) r st)
end
end eq_refl.
(* Proof obligations elided *)

Definition lifti (n : N) (t : term) (k : N) (st : state) : term * state :=
Fixm msr (lifto_rec n) (t,k,st).

```

(c) Definition of lifti

Figure 19: λ -terms revisited, using the shallow-embedding approach

roughly 4 s to sort the list L using `vm_compute`. The downside of these definitions is made apparent as soon as we start to prove that our four functions `lift`, `subst`, `hnf` and `nf` enjoy a simple correctness property: that they always return terms that are well-formed and preserve well-formedness of the global state. Proving this requires around 600 lines of code, which brings the total size of this formalization to 900 lines, before we even started to prove meta-properties like the fact that `hnf` or `nf` implement beta-reduction!

8.4 Discussion

In this section, we have demonstrated that our design patterns can be applied to other settings than BDDs. This exercise helped us to refine the presentation of smart constructors we gave in §6.1, and gave us the opportunity to present another flavor of the shallow-embedding technique, that does not rely on the mutually recursive inductives we used in §5.2.

In short, this is a nice *exercice de style*. Yet, we believe that this study is also significant by itself. Reduction in the λ -calculus is representative of the kind of computation that arises in Coq, e.g. in the conversion test. Our case study can be seen as a step stone for future works that would attempt to prove the correctness of *efficient* implementations of symbolic algorithms. On the one hand, one could use the PURE-SHALLOW approach to implement hash-consing and memoization, yet remain inside the bounds of what Coq’s extraction mechanism can safely handle. On the other hand, one could bite the bullet, use the SMART approach, and focus on more interesting details than hash-consing and memoization.

9 Conclusion

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”.

9.1 Related work

Imperative features inside the specification language. An obvious solution to implement and reason about imperative algorithms is to have these imperative features present in the modeling language of the prover. Some provers directly target high-level programming languages with data structures, references and imperative features: an example is KeY [2], which

targets a large subset of Java. While such features are not available in Coq, there are two conceptual difficulties with this approach that would have made it impractical in our case-studies. First, BDD algorithms implemented using hash-consing are functional in a high-level view: BDD operations are very clearly given functionally, by induction; but also because hash-consing is suitable only for immutable structures¹¹. It therefore seems strange to have to program them in an imperative language, furthermore one that complicates common functional idioms (e.g. pattern-matching). Second, the meta-theory for such languages is typically huge, with intricate proof rules having to deal with mutable data, references, late binding etc. It is not obvious how much we can trust the proof system with respect to the semantics of the language.

A second option is to add to an existing functional specification language certain imperative traits as *monads*, then modify the extraction function from the specification language to the target language so as to translate monadic operations into imperative calls, as has been done for Isabelle/HOL [6]; this approach has been used to verify a BDD package [8]. Special proof idioms have to be used for monadic programs, in addition to the general difficulty of programming in monadic style (see §2.1). To the best of our knowledge, this approach has not been investigated in Coq.

A third option is to integrate into the specification language some essentially imperative data structures (e.g. mutable arrays, from which hash tables can be implemented), but present them in a functional fashion (e.g. an update to a mutable array is treated as returning a new array, same as the previous except for the updated location). The imperative features are then implemented efficiently for evaluation of expressions inside the prover, and are mapped to native imperative features of the target language during extraction. For instance, an experimental version of Coq exists, with native integers and arrays [1]. Again, all difficulties with monads (see §2.1) apply, plus there is the problem of running a nonstandard version of Coq.

A fourth option is to *deeply embed* a subset of an imperative language into Coq: the programs of this imperative language are given a semantics inside Coq, and correctness properties are proved with respect to this semantics. This idea is discussed by Vafeiadis [17], but, as he remarks, this largely precludes the use of regular proof tactics: we have to develop proof steps specific to the language being embedded, and prove these steps correct with respect to the semantics.

Verification of BDD algorithms in Coq. Verma et al. [19, 18] implemented and proved correct in Coq a BDD library featuring efficient negation

¹¹Or at least for structures behaving as though they were immutable; for instance, we can perform hash-consing on a structure if the mutable information in the structure is just used for caching and does not affect the hashcode.

and disjunction; other operations like conjunction, implication and so on are implemented as derived operations. The BDDs produced are reduced and shared.

As we said, we chose to implement a fresh one because the code associated to their paper did not age well w.r.t. the evolution of Coq. Beside the fact that they investigated garbage collection, there is no conceptual difference between their library and our PURE-DEEP approach.

Verification of BDD algorithms in other theorem provers In Isabelle/HOL, Ortner and Schirmer [15] verified the implementation of a normalization algorithm for binary decision diagrams. That is, their algorithm takes as input a BDD, and outputs the corresponding ROBDD. Their formalization is built on the Burstall-Bornat memory model: they build one heap of type `ref ~> value` for each component of a BDD node, with `ref` the abstract type of memory addresses. Using a split-heap model makes it easier to reason about heap-allocated data structures in tools such as Why3. We believe however, that our formalization is more direct than theirs, and more suitable for efficient implementations in Coq.

Boyer and Hunt [4] developed an extension of ACL2 that uses hash-consing to give canonical representatives to ACL2 objects. This makes it possible to memoize some ACL2 user defined functions. As a case study, they implemented a BDD library in ACL2. Remark that this implementation is based on the fact that ACL2 exposes hash-consing primitives and the associated reasoning principles to the user. It is unclear to what extent these features could be added to the Coq proof assistant.

Hash-consing in the execution language Jean Goubault-Larrecq's HimML programming language [9, 11, 10] is an extension of core Standard ML with primitive finite set and map datatypes with a run-time system architected around the concept of maximal sharing, or systematic hash-consing.

9.2 Future work

We conclude with directions for future work. First, we believe that the smart constructors approach is generalizable to a huge variety of inductive types. One can imagine that it could be 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]. Where to stop? Should the extraction mechanism also provide builtin functional maps and sets upon hash-consed types?

Second, we look forward to investigating to what extent one could provide a certified version of the hash-consing library proposed by Conchon and Filliâtre [7].

Third, we confess that our implementations of BDDs are a bit rough. We would like to polish the implementations of our PURE-SHALLOW and SMART approaches, and release them as user-friendly libraries. In particular, we need to add some operations like quantification or functional composition.

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] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*. Vol. 4334. LNCS. Springer, 2007. ISBN: 3-540-68977-X. DOI: 10.1007/978-3-540-69061-0.
- [3] Bruno Blanchet et al. “Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software”. In: *The Essence of Computation: Complexity, Analysis, Transformation*. LNCS 2566. Springer, 2002, pp. 85–108.
- [4] Robert S. Boyer and Warren A. Hunt Jr. “Function memoization and unique object representation for ACL2 functions”. In: *ACL2*. ACM, 2006, pp. 81–89.
- [5] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux.
- [6] Lukas Bulwahn et al. “Imperative Functional Programming with Isabelle/HOL”. In: *Theorem Proving in Higher Order Logics (TPHOL)*. Vol. 5170. LNCS. Springer, 2008, pp. 134–149. ISBN: 978-3-540-71065-3. DOI: 10.1007/978-3-540-71067-7_14.
- [7] 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>.
- [8] Mathieu Giorgino and Martin Strecker. “Correctness of Pointer Manipulating Algorithms Illustrated by a Verified BDD Construction”. In: *Formal Methods (FM)*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Vol. 7436. LNCS. Springer, 2012, pp. 202–216. DOI: 10.1007/978-3-642-32759-9_18.
- [9] Jean Goubault. “HimML: Standard ML with Fast Sets and Maps”. In: *5th ACM SIGPLAN Workshop on ML and its Applications*. 1994. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.4967>.

- [10] Jean Goubault. *Implementing Functional Languages with Fast Equality, Sets and Maps: an Exercise in Hash Consing*. Tech. rep. Bull S.A. Research Center, May 1994. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.1757>.
- [11] Jean Goubault-Larrecq. *HimML: HimML is a map-oriented ML*. Dec. 2008. URL: <http://www.lsv.ens-cachan.fr/~goubault/himml-dwnld.html>.
- [12] Gérard Huet. “Constructive Computation Theory”. Course notes, available at <http://yquem.inria.fr/~huet/PUBLIC/CCT.pdf>.
- [13] Donald E. Knuth. “The Art of Computer Programming”. In: vol. 4A. Binary decision diagrams. Addison-Wesley, 2011. Chap. 7.1.4. ISBN: 978-0201038040.
- [14] Guillaume Melquiond. “Floating-point arithmetic in the Coq system”. In: *Information and Computation* 216 (2012), pp. 14–23. DOI: 10.1016/j.ic.2011.09.005.
- [15] Veronika Ortner and Norbert Schirmer. “Verification of BDD Normalization”. In: *TPHOLs*. Vol. 3603. Lecture Notes in Computer Science. Springer, 2005, pp. 261–277.
- [16] *The Why3 platform 0.80*. URL: <http://why3.lri.fr>.
- [17] Viktor Vafeiadis. “Adjustable References”. In: *ITP*. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 328–337.
- [18] Kumar Neeraj Verma and Jean Goubault-Larrecq. *Reflecting BDDs in Coq*. Anglais. Rapport de recherche RR-3859. INRIA, 2000. HAL: [inria-00072797](https://hal.inria.fr/inria-00072797).
- [19] 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. DOI: 10.1007/3-540-44464-5_13.