

# BDDs in Coq

*A thesis in partial completion of the requirements for the degree of*

**Bachelor of Technology**

*in*

**Computer Science and Engineering**

*by*

**Kumar Neeraj Verma  
Entry Number 96139**

*under supervision of*

**Dr Sanjiva Prasad  
*and***

**Dr S Arun-Kumar**



**Department of Computer Science and Engineering  
Indian Institute of Technology, Delhi  
2000**

# Certificate

This is to certify that this thesis titled **BDDs in Coq** submitted by Kumar Neeraj Verma towards partial completion of the requirements for the degree of Bachelor of Technology in Computer Science & Engineering is a record of bona-fide work done by him under our supervision.

This work has not been submitted to any other institute or university for award of any degree or diploma.

**Dr Sanjiva Prasad**

**Dr S Arun-Kumar**

## Acknowledgement

I am grateful to Dr S Arun-Kumar, Dr Sanjiva Prasad and Dr Jean Goubault-Larrecq for guiding me in this project.

**Kumar Neeraj Verma**

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Model Checking and Theorem Proving . . . . .	4
1.2	Previous Work . . . . .	4
1.3	Need for Garbage Collection . . . . .	5
1.4	Objectives . . . . .	5
1.5	Organization of the Report . . . . .	5
<b>2</b>	<b>Binary Decision Diagrams (BDDs)</b>	<b>7</b>
<b>3</b>	<b>Coq</b>	<b>10</b>
<b>4</b>	<b>The Map Library</b>	<b>12</b>
<b>5</b>	<b>Changes in Representation and Invariants</b>	<b>14</b>
<b>6</b>	<b>The Garbage Collection Routines</b>	<b>17</b>
6.1	Requirements on the garbage collector . . . . .	17
6.2	The Mark Phase . . . . .	19
6.3	The First Sweep Phase . . . . .	22
6.4	Cleaning of Sharing and Memoization Maps . . . . .	29
6.5	Combining Everything . . . . .	35
<b>7</b>	<b>Integration of the Garbage Collector into the BDD algorithms</b>	<b>37</b>
7.1	Allocation . . . . .	38
7.2	Logical Operations . . . . .	40
<b>8</b>	<b>Experiments</b>	<b>44</b>
<b>9</b>	<b>Conclusion</b>	<b>47</b>

# Chapter 1

## Introduction

### 1.1 Model Checking and Theorem Proving

Binary Decision Diagrams (BDDs) are compact and canonical representations of propositional logic formulas as graphs, and are widely used in modern model checking tools. Model checking allows fast and automatic verification of finite state systems. Use of BDDs in symbolic model checkers has allowed very large verification problems to be solved.

Theorem proving is another approach to formal verification where the specification of a system is proved as a theorem. While use of model checking techniques is limited to verification of finite state systems, theorem provers can be used for verification of systems with infinite number of states. Theorem provers have the advantage of being expressive and we can use them to prove general properties of large (possibly infinite) classes of systems. As a simple example we can use a theorem prover to prove correctness of an  $n$  bit adder for arbitrary  $n$ . Model checkers on the other hand allow verification of only one system at a time. Theorem provers allow us to use proof techniques like induction etc. We can use modularity by proving properties of different components of a system and then combining them to obtain properties of composite systems.

However, theorem provers are not automated, and the proofs need to be written by the user, which involves a considerable amount of effort. Therefore combining the techniques of model checking and theorem proving seems to be a good approach to this problem. This can be done, for example, by implementing the model checker in a proof assistant like Coq and proving the correctness of this implementation. This will allow the results of the model checker to be used directly inside the theorem prover.

### 1.2 Previous Work

This work was a continuation of a previous implementation of BDDs in Coq [VGL00] [Ver99], which is a proof assistant. This implementation contained a representation of BDDs as suitable data structures in Coq. The invariants of this representation as well as their semantics were defined. The basic logical operations on BDDs (negation, disjunction, conjunction, implication, double implication) were implemented and their correctness was formally proved in

terms of the semantics, using the invariants. Finally a tautology checker for boolean expressions was implemented which worked by building a BDD for the expression. Its correctness was proved. The uniqueness of BDDs for this representation was proved as well.

### 1.3 Need for Garbage Collection

The problem with the previous implementation was that the mechanism for adding new nodes into the configuration was very crude. It did not provide any mechanism for removing nodes from the configuration that are no longer required. Having unused nodes in the configuration is a problem because BDDs tend to grow very large. Even formulas with very small sized BDDs can involve much larger BDDs during their construction. This limitation makes this implementation impractical for any serious use. The unused nodes in the configuration would soon use up all the available memory. For any serious application of BDDs (like model checking, which is the main application of BDDs), it is necessary to continuously remove unwanted BDDs from the configuration.

### 1.4 Objectives

The aim of the project is to modify the previous implementation of BDDs in Coq to incorporate garbage collection of unused nodes. This involves changing the existing representation and then implementing the garbage collection routines. Besides the existing mechanism for allocation of nodes needs to be modified, and the logical operations need to be modified to enable the garbage collector to be called automatically.

All this of course needs to be formally proved correct. Besides proving the correctness of the garbage collector, we need to prove that the other logical operations work correctly, in presence of the garbage collector. This involves making suitable modifications in the invariants and redoing the previous proofs for the new invariants.

### 1.5 Organization of the Report

Chapter 2 gives an introduction to BDDs. In chapter 3 we give a brief description of Coq. We then describe the map library in chapter 4, which is used in our implementation. In chapter 5 we describe the changes in the previous representation of BDDs and the invariants on the representation. In chapter 6 we give the main garbage collection algorithms and proofs, and in chapter 7 describe how the garbage collector was integrated into the rest of the BDD algorithms. We give the Coq code for the definitions of the main functions. We also give the Coq code for the statements of the important theorems proved. All these theorems have been formally proved in Coq (i.e. the proofs have been mechanically checked by Coq), so they can be safely assumed to be correct. However instead of listing out the proof scripts, which would have been too bulky and unreadable, we give an outline of the proofs. We explain the relevant portions of the old implementation which are necessary for understanding the present implementation. Complete details of the previous implementation can be obtained

from the report [VGL00]. A few experiments are presented in chapter 8. We finally conclude in chapter 9.

## Chapter 2

# Binary Decision Diagrams (BDDs)

BDDs are compact and canonical representations of boolean functions. They have been found to be very efficient for model checking applications and are widely used by modern model checkers. Verification problems involving very large sets of states have been solved using them.

BDDs represent a boolean expression as a directed acyclic graph. This representation is based on the Shannon decomposition of any boolean expression  $\phi$  in the form *if  $x$  then  $\phi[x := \mathbf{1}]$  else  $\phi[x := \mathbf{0}]$* , where  $x$  is a variable. Any boolean expression can be written using only the terms  $\mathbf{1}$  and  $\mathbf{0}$  and the *if then else* construct. This gives us a decision tree whose leaf nodes are  $\mathbf{1}$  and  $\mathbf{0}$  and the internal nodes contain variables. The notation  $x \longrightarrow \phi_1; \phi_2$  is used for the expression *if  $x$  then  $\phi_1$  else  $\phi_2$* .

BDDs are optimized versions of decision trees and represent a formula as a directed acyclic graph instead of a tree. The techniques used in this representation are

- **Sharing** : all identical nodes (i.e. containing the same variable and having same successors) are merged. Thus there is no duplication of nodes.
- **Reduction** : a node having two identical children nodes represents a redundant decision and can be removed. All links to this node are replaced by links to the child.
- **Ordering** : An ordering on variables is used, and variables on all paths in the BDD occur in that order.

The above three conditions ensure that BDDs are canonical representations of boolean functions upto graph isomorphism. Thus if two propositional formula are equivalent, then the corresponding BDDs would be identical. In particular, any tautology is represented just by the leaf node  $\mathbf{1}$ . Any contradiction is represented by  $\mathbf{0}$ .

The logical operations on BDDs like  $\neg$ ,  $\vee$  etc. work by a straight forward recursion on the structure of the BDDs. They use the orthogonality property of BDDs.

$$\neg (x \longrightarrow \phi_1; \phi_2) = x \longrightarrow \neg \phi_1 ; \neg \phi_2$$

$$(x \longrightarrow \phi_1; \phi_2) \vee (x \longrightarrow \psi_1; \psi_2) = x \longrightarrow (\phi_1 \vee \psi_1) ; (\phi_2 \vee \psi_2)$$

$$(x \longrightarrow \phi_1; \phi_2) \vee \phi = x \longrightarrow (\phi_1 \vee \phi) ; (\phi_2 \vee \phi)$$

For  $\vee$  we have different cases according to whether the variable at the root of the two BDDs are same or different.

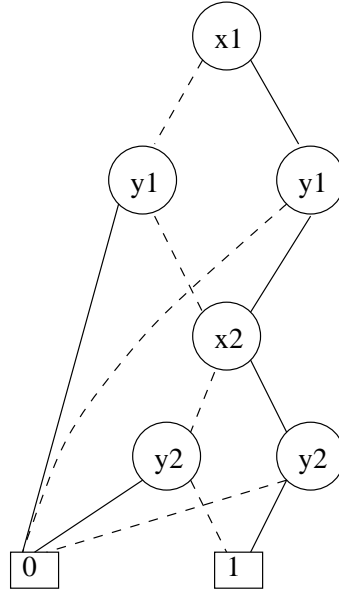


Figure 2.1: A BDD for  $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$  with ordering  $x_1 > y_1 > x_2 > y_2$

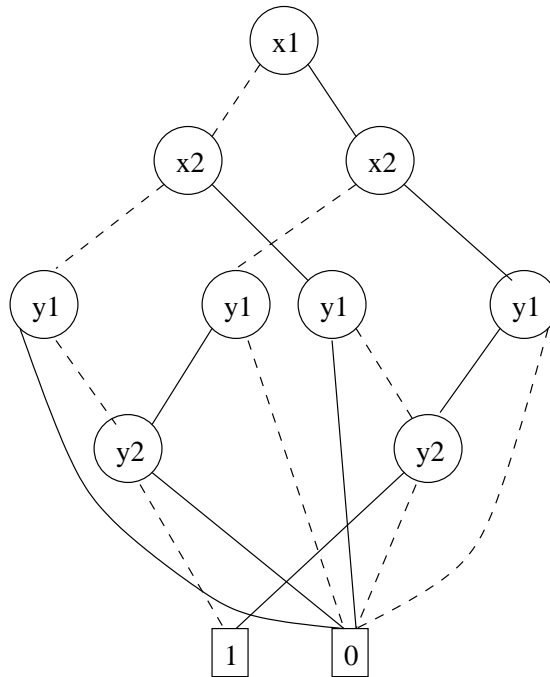


Figure 2.2: A BDD for  $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$  with ordering  $x_1 > x_2 > y_1 > y_2$

While implementing BDDs, the standard technique is to have a global memory in which we store all the BDDs to be used and all identical subgraphs of different BDDs are shared. So checking equivalence of formulas reduces to checking equality of the addresses where the corresponding BDDs are stored, instead of checking isomorphism of the two BDDs.

# Chapter 3

## Coq

We give here a very brief introduction to Coq and explain some notation. See [BBC<sup>+</sup>99] for a formal and complete description or [HKPM98] for an introduction.

Coq is a proof assistant based on the Calculus of Inductive Constructions. It allows interactive development of formal proofs which are then mechanically checked by Coq. Coq can also be considered as a programming language and can be used for development of programs. Both programs and proofs are represented in Coq as lambda terms. However Coq provides an interface for interactive development of proofs by breaking the required goals into subgoals using a set of tactics.

Even a person not familiar with Coq should be able to understand the Coq code for most of the definitions and statements of theorems in the report. We clarify here some of the notation of Coq.

We have two basic sorts `Set` and `Prop` in Coq which are respectively the types of mathematical sets (like `nat` and `bool`) and that of the mathematical propositions or formulas. If  $F$  is any proposition,  $(x:T)F$  is also a proposition, read as, ‘for all  $x$  of type  $T$ ,  $F$ ’.  $(\exists x : T \mid F)$  is a proposition read as ‘exists  $x$  of type  $T$ , such that  $F$ ’.

Coq provides mechanism for inductively defined sets, predicates, relations, etc.

For example the set `nat` is defined in the standard libraries as follows using the constructor `S` for the successor function.

```
Inductive nat : Set := O : nat | S : nat->nat.
```

The notation  $[x:T]f$  is used for a function with formal argument  $x$  of type  $T$  and body  $f$ .

The addition function of `nat` can be defined as the following fixpoint definition.

```
Fixpoint plus [n:nat] : nat -> nat :=  
  [m:nat]Cases n of  
    O    => m  
  | (S p) => (S (plus p m)) end.
```

The usual order on `nat` can be inductively defined using two constructors as follows.

```
Inductive le [n:nat] : nat -> Prop
  := le_n : (le n n)
  | le_S : (m:nat)(le n m)->(le n (S m)).
```

The first clause says that  $n \leq n$  and the second says that for any  $m$  if  $n \leq m$  then  $n \leq m + 1$ .

All examples in this chapter have been taken from the standard library distributed with `coq`.

# Chapter 4

## The Map Library

The implementation makes extensive use of the map library [GL98] which is a part of the user contributions to Coq. We describe here the main definitions and lemmas from the library that are used. Unfortunately no documentation of the implementation is available as of now.

The map library provides a representation for finite maps from addresses to any given set  $A$ . Finite sets of addresses are represented as functions from  $ad$  to the `unit` type which has only one element. Addresses belong to the type  $ad$  which is a binary representation of integers.

```
Inductive ad : Set := ad_z : ad | ad_x : positive->ad.
```

$ad\_z$  represents the integer 0 and  $(ad\_x\ p)$  represents the positive integer  $p$ . Positive integers are defined in the `ZArith` module of the standard Coq library.

```
Inductive positive : Set :=
  xI : positive->positive | x0 : positive->positive | xH : positive.
```

$xH$  represents the integer 1,  $(x0\ p)$  represents the integer  $p$  followed by a 0 bit (in the binary representation) and  $(xI\ p)$  represents the integer  $p$  followed by a 1 bit.

Maps are defined as

```
Inductive Map [A:Set] : Set :=
  M0 : (Map A)
  | M1 : ad->A->(Map A)
  | M2 : (Map A)->(Map A)->(Map A).
```

For any set  $A$ ,  $(Map\ A)$  represents a function from a finite set of addresses (represented by the set  $ad$ ) to the set  $A$ .  $(M0\ A)$  represents the empty map,  $(M1\ A\ a\ y)$  is the singleton map mapping the address  $a$  to the element  $y$ . Maps with more than two addresses in the domain are stored in the form  $(M2\ A\ m1\ m2)$  where  $m1$  and  $m2$  are maps. All the elements are stored according to the binary representation of the addresses. We have the functions `MapGet` and `MapPut` for respectively looking up and making insertions into the map. `MapGet` looks up a map for an address and returns either  $(SOME\ A\ x)$ , if  $x$  is stored at the address, or  $(NONE\ A)$  if nothing is stored at the address. These belong to the type `(option A)`.

```

Inductive option [A:Set] : Set :=
  NONE : (option A) | SOME : A->(option A).

```

```

MapGet : (A:Set)(Map A)->ad->(option A)

```

```

MapPut : (A:Set)(Map A)->ad->A->(Map A)

```

The semantics of `MapPut` is given by `MapPut_semantics`, which says that if we put an element `x` at an address `a` in a map, then looking up the map for the address `a` will return the element `x` (anything else previously stored there is replaced). Looking up for any other address returns the same value as the original map.

```

MapPut_semantics : (A:Set; m:(Map A); a:ad; y:A)
  (eqm A (MapGet A (MapPut A m a y))
   [a':ad](if (ad_eq a a') then (SOME A y) else (MapGet A m a'))))

```

`(eqm A)` is the equality of two functions from `ad` to `(option A)`. Here, by function, we mean functions in the sense of `Coq`, and not the maps described here.

We also use some other functions like `MapDomRestrTo` which restricts the domain of a map to the domain of another map.

```

MapDomRestrTo : (A,B:Set)(Map A)->(Map B)->(Map A)

```

```

MapDomRestrTo_semantics : (A,B:Set; m:(Map A); m':(Map B))
  (eqm A (MapGet A (MapDomRestrTo A B m m'))
   [a0:ad] Cases (MapGet B m' a0) of
     NONE => (NONE A)
     | (SOME _) => (MapGet A m a0)
   end)

```

# Chapter 5

## Changes in Representation and Invariants

The previous configuration consisted of the following components.

- `BDDstate` which stored all the nodes in the configuration. It was represented as a map from addresses to tuples of a variable and two other addresses (representing the children BDDs)
- `BDDsharing_map` which represented the inverse map of the `BDDstate`
- a counter which represented the last address where a node is stored
- memoization maps for conjunction and disjunction, which store previously computed results

New nodes were added into the configuration by simply storing it at the address pointed to by counter and incrementing the counter.

To allow garbage collection we need to keep additional information about each node, whether it is free or not. For this, we added a free list to the configuration, represented by the `List` type.

### Definition 1

Definition `BDDfree_list := (list ad)`.

The other option was to implement it as a map, or to keep an additional bit with each node in the `BDDstate` to indicate whether it is free or not. However the advantage of having the list representation is that allocation is simplest. The first address in the list is allocated.

The free list stores all those addresses between one and the counter which are not in the `BDDstate`. This is stated in by the predicate `BDDfree_list_OK`.

**Definition 2** *The free list in the BDD configuration is OK if it does not contain any duplicates, and it contains exactly exactly those addresses which are greater than one (since the addresses zero and one are reserved for the leaf nodes), less than counter, and are not in the `BDDstate`.*

```

Definition BDDfree_list_OK := [bs:BDDstate; fl:BDDfree_list; counter:ad]
  (no_dup_list ? fl)
  /\ ( (node:ad)(In node fl)
      <->(ad_le (ad_x (x0 xH)) node)=true
      /\ (ad_le (ad_S node) counter)=true
      /\ (MapGet ? bs node)=(NONE ?) ).

```

(no\_dup\_list ? fl) means that the list fl does not contain duplicates. This is required so that the allocation function works fine.

### Definition 3

```

Inductive no_dup_list [A:Set] : (list A) -> Prop :=
  no_dup_nil : (no_dup_list ? (nil A))
| no_dup_cons : (a:A; l:(list A))~(In a l) -> (no_dup_list ? l)
  -> (no_dup_list ? (cons a l)).

```

Besides this we add another list to the configuration which stores all those BDDs of the configuration which the user requires and should not be removed by the garbage collector if it is called. The list contains the addresses of the root nodes of these BDDs.

The only invariant on this list is that all the nodes in it should be present in the BDDstate.

### Definition 4

```

Definition used_list_OK_bs := [bs:BDDstate; ul:(list ad)]
  (node:ad)(In node ul)
  -> (node_OK bs node).

```

The node\_OK predicate is as in the previous implementation, and it represents those addresses which are present in the BDDstate, i.e. which are not free.

The invariants for the rest of the configuration (BDDstate, BDDsharing\_map, counter, memoization maps) remain more or less the same. The conjunction of all these gives the conditions when a configuration is OK.

### Definition 5

```

Definition BDDconfig := BDDstate * BDDsharing_map * BDDfree_list * ad
  * BDDneg_memo * BDDor_memo.

```

The ad in the definition of BDDconfig refers to the counter. There is a minor difference in the way the configuration has been defined, from the way it was previously defined. We have included the memoization maps in the definition of the configuration, instead of passing it separately in all the functions. This is just for convenience and does not affect the functions and proofs much. It makes the definitions of functions and the statements of lemmas look smaller. The invariants on the memoization maps get included in the invariants on the whole configuration.



# Chapter 6

## The Garbage Collection Routines

We have implemented a mark and sweep garbage collector, since it seems to be the simplest to implement and prove correct. The reference counting scheme is more complex and would have been difficult to prove correct, requiring complex invariants to be carried around with the configuration.

The garbage collector takes as input the configuration and a list of nodes representing the BDDs that are required by the user. The following are the main steps in the garbage collection algorithm.

- Mark phase in which all the nodes in the `BDDstate` reachable from the nodes in the used list are marked using a depth first search.
- Sweep phase in which all the unmarked nodes from the `BDDstate` are removed and added to the free list.
- Cleaning of the sharing and memoization maps. All entries in these maps referring to nodes which have been freed are removed.

### 6.1 Requirements on the garbage collector

In order to be able to prove the correctness of the BDD algorithms, we need to identify some properties that must be proved about the garbage collector. We have followed a modular approach by separating the implementation of the garbage collector from the rest of the BDD algorithms. We specify some general requirements that must be satisfied by any garbage collector. The rest of the BDD implementation as well as the proofs work by assuming any given garbage collector that satisfies these requirements. This gives us the freedom to change the garbage collector at any time without changing the rest of the implementation. We would just need to prove that the new garbage collector satisfies all the requirements.

The type of a garbage collector is `BDDconfig -> (list ad) -> BDDconfig`.

**Definition 8** *A garbage collector is OK if the configuration returned by it is OK and does not contain any new nodes (which were not in the old configuration), and if it preserves all the nodes reachable from the nodes in the input list.*

```

Definition gc_OK := [gc:(BDDconfig -> (list ad) -> BDDconfig)]
  (cfg:BDDconfig; ul:(list ad))
  (BDDconfig_OK cfg)
  -> (used_list_OK cfg ul)
  -> (BDDconfig_OK (gc cfg ul))
  /\ (used_nodes_preserved cfg (gc cfg ul) ul)
  /\ (no_new_node cfg (gc cfg ul)).

```

(no\_new\_node bs bs') means that all nodes in bs' are also in bs.

### Definition 9

```

Definition no_new_node_bs := [bs,bs':BDDstate]
  (x:BDDvar; l,r,node:ad)
  (MapGet ? bs' node)=(SOME ? (x,(l,r)))
  -> (MapGet ? bs node)=(SOME ? (x,(l,r))).

```

While the first condition in the definition of `gc_OK` (that of preservation of used nodes) is an obvious requirement for any garbage collector, the need for the second condition (`no_new_node`) is explained later in section 7.1.

Note that we have not put any condition like the garbage collector should remove all unwanted nodes from the configuration. We would obviously require any good garbage collector to do that. However we have put only those conditions here which are sufficient for the proofs of the rest of the BDD algorithms to go through. But it does not stop us from implementing a garbage collector which satisfies those requirements.

The definition of the relation `used_nodes_preserved` makes use of a few preliminary definition. The following definition formalizes the notion of when a node is reachable from another node.

### Definition 10

```

Inductive nodes_reachable [bs:BDDstate] : ad -> ad -> Prop :=
  nodes_reachable_0 : (node:ad)(nodes_reachable bs node node)
| nodes_reachable_1 : (node,node',l,r:ad)(x:BDDvar)
  (MapGet ? bs node)=(SOME ? (x,(l,r)))
  -> (nodes_reachable bs l node')
  -> (nodes_reachable bs node node')
| nodes_reachable_2 : (node,node',l,r:ad)(x:BDDvar)
  (MapGet ? bs node)=(SOME ? (x,(l,r)))
  -> (nodes_reachable bs r node')
  -> (nodes_reachable bs node node').

```

The first clause says that the reachability relation is reflexive. The second and third clauses say that that all nodes reachable from the left or right child of a node  $n$  are also reachable from  $n$ .

## Definition 11

```
Definition node_preserved_bs := [bs,bs':BDDstate; node:ad]
  (x:BDDvar; l,r,node':ad)
  (nodes_reachable bs node node')
  -> (MapGet ? bs node')=(SOME ? (x,(l,r)))
  -> (MapGet ? bs' node')=(SOME ? (x,(l,r))).
```

```
Definition node_preserved :=
  [cfg,cfg':BDDconfig](node_preserved_bs (Fst cfg) (Fst cfg')).
```

(node\_preserved cfg cfg' node) means that all nodes reachable from node in the configuration cfg are also present in cfg'.

Finally we define the relation used\_nodes\_preserved which says that all nodes present in the list of used nodes are preserved.

## Definition 12

```
Definition used_nodes_preserved_bs := [bs,bs':BDDstate; ul:(list ad)]
  (node:ad)(In node ul)
  -> (node_preserved_bs bs bs' node).
```

```
Definition used_nodes_preserved
:= [cfg,cfg':BDDconfig](used_nodes_preserved_bs (Fst cfg) (Fst cfg')).
```

## 6.2 The Mark Phase

In the mark phase, we compute the set of nodes reachable from the nodes in the used list. This is represented as a map. This is done by considering the nodes in the list one at a time, and adding all nodes reachable from it to a set. The following function computes the set of nodes reachable from a single node and adds it to another (input) set.

## Definition 13

```
Fixpoint add_used_nodes_1
  [bs:BDDstate; node:ad; marked:(Map unit); bound:nat] : (Map unit) :=
  Cases bound of
  0 => (* Error *) (M0 unit)
  | (S bound') =>
    Cases (MapGet ? marked node) of
    NONE =>
      Cases (MapGet ? bs node) of
      NONE => marked
      | (SOME (x,(l,r))) =>
```

```

      (MapPut ? (add_used_nodes_1 bs r
                (add_used_nodes_1 bs l marked bound')
                bound') node tt)
    end
  | (SOME tt) => marked
end
end.

```

The function first checks whether `node` is already present in the set `marked`, and if so, returns the same set. This is to ensure that the same subtree is not traversed more than once. Otherwise it recursively adds first the nodes reachable from the left child, and then the nodes reachable from the right child. Finally the node `node` is added to the set.

The last input `bound` is a trick which was also used in the previous implementation to get around the problem that coq accepts only recursive definitions with structural recursion on the last argument. This is to ensure that only terminating functions are accepted by coq. BDDs being represented using maps, there is no direct way of recursing on the structure of the BDDs. So we recurse on the argument `bound` of type `nat`. While calling the function, we need to ensure that the argument `bound` is greater than the maximum possible depth of recursion. This is easy to calculate by looking at the variable stored at the root of the BDD. This is done by the following function.

#### Definition 14

```

Definition add_used_nodes := [bs:BDDstate; node:ad; marked:(Map unit)]
  (add_used_nodes_1 bs node marked (S (nat_of_ad (bs_var' bs node)))).

```

For `add_used_nodes`, we need to prove that it adds exactly those nodes which are reachable from the input node. We first prove it for the function `add_used_nodes_1`. We require first the following auxiliary lemma.

**Lemma 1** *add\_used\_nodes\_1 retains all nodes which were originally present in the set.*

```

Lemma add_used_nodes_1_lemma_1 :
  (bound:nat; bs:BDDstate; node:ad; marked:(Map unit))
  (BDDstate_OK bs)
  -> (lt (nat_of_ad (bs_var' bs node)) bound)
  -> (node':ad)
  (in_dom ? node' marked)=true
  -> (in_dom ? node' (add_used_nodes_1 bs node marked bound))=true.

```

**Proof:** The proof is using induction on `bound`. Note that this has been assumed to be greater than the quantity `bs_var'` for the input node. This ensures that `bound` is greater than the maximum possible depth of recursion.  $\square$

We now prove the required property for `add_used_nodes_1`. For that we require a condition on the input set `marked`. It should contain all nodes reachable from any node in it. This condition is defined below.

**Definition 15** *A set of nodes is called closed if it contains all nodes reachable from any node in it.*

```

Definition set_closed := [bs:BDDstate; marked:(Map unit)]
  (node,node':ad)
  (in_dom ? node marked)=true
  -> (nodes_reachable bs node node')
  -> (in_dom ? node' bs)=true
  -> (in_dom ? node' marked)=true.

```

This condition is required because if some node is already present in the set then we do not look at the children of that node.

**Lemma 2** *The set returned by `add_used_nodes_1` is closed and it contains all those and only those nodes which were either present in the input set or are reachable from some node in the input list. The assumption is that the input set is closed.*

```

Lemma add_used_nodes_1_lemma_2 :
  (bound:nat; bs:BDDstate; node:ad; marked:(Map unit))
  (BDDstate_OK bs)
  -> (lt (nat_of_ad (bs_var' bs node)) bound)
  -> (set_closed bs marked)
  -> ( (node':ad)
    ( (nodes_reachable bs node node')
      /\ (in_dom ? node' bs)=true )
    -> (in_dom ? node' (add_used_nodes_1 bs node marked bound))=true )
  /\ ( (node':ad)
    (in_dom ? node' (add_used_nodes_1 bs node marked bound))=true
    -> ( (in_dom ? node' marked)=true
      \/ ( (in_dom ? node' bs)=true
        /\ (nodes_reachable bs node node') ) ) )
  /\ (set_closed bs (add_used_nodes_1 bs node marked bound)).

```

**Proof:** Using induction on `bound`. We prove the three conjuncts in the conclusion together in a single lemma because we require all of them in the induction hypothesis.  $\square$

These lemmas are then used to prove the corresponding properties for `add_used_nodes`.

The function `mark` defined below computes the nodes reachable from all the nodes in the used list, by calling the function `add_used_nodes` for each node in the list.

**Definition 16**

```

Definition mark := [bs:BDDstate; used : (list ad)]
  (fold_right (add_used_nodes bs) (MO unit) used).

```

**Lemma 3** *The set returned by mark contains exactly those nodes which are reachable from some node in the input list.*

```

Lemma mark_lemma_2 : (bs:BDDstate; used:(list ad))
  (BDDstate_OK bs)
  -> (node:ad)
    (in_dom ? node (mark bs used))=true
    <-> (EX node':ad | (In node' used)
        /\ (nodes_reachable bs node' node)
        /\ (in_dom ? node bs)=true).

```

**Proof:** Follows immediately from the following auxiliary lemma. □

**Lemma 4**

```

Lemma mark_lemma_1 :
  (used:(list ad); bs:BDDstate)
  (BDDstate_OK bs)
  -> (set_closed bs (fold_right (add_used_nodes bs) (M0 unit) used))
  /\ (node:ad)
    (in_dom ? node (fold_right (add_used_nodes bs) (M0 unit) used))=true
    <-> (EX node': ad | (In node' used)
        /\ (nodes_reachable bs node' node)
        /\ (in_dom ? node bs)=true).

```

**Proof:** The statement of this lemma is same as that of the previous one except that we have added the extra conclusion that the output set is closed. This is required to be in the induction hypothesis. Also the definition of mark using fold\_right has been expanded. The proof is using induction on the list used. □

## 6.3 The First Sweep Phase

In this phase, we compute the new BDDstate by keeping only the nodes that were marked in the previous phase. This can be easily done by the function MapDomRestrTo from the map library, which restricts the domain of a map (in this case, the BDDstate) to the domain of another map (in this case the set of marked nodes).

**Definition 17**

```

Definition new_bs := [bs:BDDstate; used:(list ad)]
  (MapDomRestrTo ? ? bs (mark bs used)).

```

The properties of the new BDDstate are proved using the lemma MapDomRestrTo\_semantics proved in the map library. The following things need to be proved about the new BDDstate as required by the conditions in gc\_OK.

- The new BDDstate is OK.
- It does not contain any new node which were not present in the old BDDstate.
- All the used nodes from the old BDDstate are preserved.

We have the following definition of the used nodes in the configuration which should be preserved. It consists of the nodes which are reachable from some node in the used list.

### Definition 18

```
Definition used_node_bs := [bs:BDDstate; ul:(list ad); node:ad]
  (EX node':ad | (In node' ul) /\ (nodes_reachable bs node' node)).
```

The following two lemmas `new_bs_lemma_1` and `new_bs_lemma_2` follow immediately from the semantics of `MapDomRestrTo` and the properties proved about the set of marked nodes.

**Lemma 5** *All used nodes are included in the new BDDstate.*

```
Lemma new_bs_lemma_1 : (bs:BDDstate; used:(list ad))
  (BDDstate_OK bs)
  -> (node:ad)(used_node_bs bs used node)
  -> (MapGet ? bs node)=(MapGet ? (new_bs bs used) node).
```

**Lemma 6** *Only used nodes are included in the new BDDstate.*

```
Lemma new_bs_lemma_2 : (bs:BDDstate; used:(list ad))
  (BDDstate_OK bs)
  -> (node:ad)(in_dom ? node (new_bs bs used))=true
  -> (used_node_bs bs used node).
```

From these we can immediately prove the second and the third requirements for the new BDDstate stated above.

**Lemma 7** *The new BDDstate does not contain any node which was not present in the old BDDstate.*

```
Lemma no_new_node_new_bs : (bs:BDDstate; used:(list ad))
  (BDDstate_OK bs) -> (no_new_node_bs bs (new_bs bs used)).
```

**Lemma 8** *All used nodes from the old BDDstate are preserved in the new BDDstate.*

```
Lemma new_bs_used_nodes_preserved :
  (bs:BDDstate; used:(list ad); node:ad)
  (BDDstate_OK bs)
  -> (used_node_bs bs used node)
  -> (node_preserved_bs bs (new_bs bs used) node).
```

Next we need to prove that the new `BDDstate` is OK, i.e, it satisfies all the invariants stated for the `BDDstate`. The following is the definition of the `BDDstate_OK` predicate from the previous implementation (with minor modifications).

```

Inductive BDDbounded [bs:BDDstate] : ad -> BDDvar -> Prop :=
  BDDbounded_0 : (n:BDDvar) (BDDbounded bs BDDzero n)
| BDDbounded_1 : (n:BDDvar) (BDDbounded bs BDDone n)
| BDDbounded_2 : (node:ad) (n:BDDvar) (x:BDDvar) (l,r:ad)
  (MapGet ? bs node)=(SOME ? (x, (l, r))) ->
  (BDDcompare x n)=INFERIEUR ->
  (ad_eq l r)=false ->
  (BDDbounded bs l x) ->
  (BDDbounded bs r x) -> (BDDbounded bs node n).

```

```

Definition BDD_OK := [bs:BDDstate; node:ad]
Cases (MapGet ? bs node) of
  NONE => (node=BDDzero) /\ (node=BDDone)
| (SOME (n, _)) => (BDDbounded bs node (ad_S n))
end.

```

```

Definition BDDstate_OK := [bs:BDDstate]
(MapGet ? bs BDDzero)=(NONE ?) /\
(MapGet ? bs BDDone)=(NONE ?) /\
(a:ad) (in_dom ? a bs)=true -> (BDD_OK bs a).

```

The `BDDbounded` predicate states the conditions when a node represents a well formed BDD. A leaf node is bounded by any variable. A node containing the variable  $x$  and having left and right children  $l$  and  $r$  is bounded by the variable  $n$  if  $x$  is less than  $n$ ,  $l$  and  $r$  are distinct and the two children are bounded by  $x$ .

The first two conditions in `BDDstate_OK` say that there is nothing stored at the addresses zero and one. These follow from `new_bs_lemma_1` and `new_bs_lemma_2`.

**Lemma 9** *There is nothing stored at the addresses zero and one in the new `BDDstate`.*

```

Lemma new_bs_zero : (bs:BDDstate; used:(list ad))
  (BDDstate_OK bs)
-> (in_dom ? BDDzero (new_bs bs used))=false.

```

```

Lemma new_bs_one : (bs:BDDstate; used:(list ad))
  (BDDstate_OK bs)
-> (in_dom ? BDDone (new_bs bs used))=false.

```

For proving the third condition in `BDDstate_OK` for the new `BDDstate_OK`, we need to prove that any non leaf node in the new `BDDstate_OK` satisfies the `BDDbounded` predicate.

**Lemma 10** *Any node in the new BDDstate which is bounded in the old BDDstate by the variable  $x$  is also bounded in the new BDDstate by the  $x$ .*

```

Lemma new_bsBDDbounded_1 :
  (n:nat; bs:BDDstate; used:(list ad); node:ad; x:BDDvar)
  (BDDstate_OK bs)
  -> n=(nat_of_ad x)
  -> (in_dom ? node (new_bs bs used))=true
  -> (BDDbounded bs node x)
  -> (BDDbounded (new_bs bs used) node x).

```

**Proof:** We have included an extra variable  $n$  in the statement of the lemma to enable us to induct on it. The proof is by well founded induction on  $n$  (which is equal to the variable  $x$ ). The case where  $node$  is a leaf node is simple because they are bounded by any variable. Now we consider the case where  $node$  contains a variable  $x_0$  and has children  $l$  and  $r$ . From `new_bs_lemma_1` and `new_bs_lemma_2`, it follows that in the older BDDstate too,  $node$  must be having the same variable and children. But since it was bounded by  $x$  so  $x_0$  must be less than  $x$ . Also  $l$  and  $r$  must be distinct. Finally we must show that  $l$  and  $r$  are bounded in the new BDDstate by  $x_0$ . In the case where they are leaf nodes, it is simple. In case they are internal nodes we use the induction hypothesis. For that we must show that  $x_0$  is less than  $x$  (already shown), the children nodes are in the domain of the new BDDstate (follows from `new_bs_lemma_1` and `new_bs_lemma_2` and the fact they are reachable from  $node$ ), and that they were bounded by  $x_0$  in the old BDDstate (this follows from the fact that  $node$  was bounded in the old BDDstate).  $\square$

Combining the above results, we get that the new BDDstate is OK.

### Lemma 11

```

Lemma new_bs_OK : (bs:BDDstate; used:(list ad))
  (BDDstate_OK bs) -> (BDDstate_OK (new_bs bs used)).

```

The nodes in the old BDDstate that are not marked in the mark phase are useless and can be put into the free list. Thus the set of nodes that are to be freed is the complement of the set of nodes from the old BDDstate that were retained in the new BDDstate. This is achieved by the function `list_of_MapDomRestrByDom1` which takes two maps  $m$  and  $m'$  and computes the set of addresses in the domain of  $m$  that are not in the domain of  $m'$ . It adds these nodes to a list of nodes which is taken as input.

### Definition 19

```

Fixpoint list_of_MapDomRestrByDom1 [pf:ad->ad; l:(list ad); m:(Map A)]
  : (Map B) -> (list ad) :=
  Cases m of
  M0 => [_:(Map B)]l
  | (M1 a y) => [m':(Map B)] Cases (MapGet B m' a) of
    NONE => (cons (pf a) l)

```

```

        | _ => l
      end
    | (M2 m1 m2) => [m':(Map B)]
      Cases m' of
        M0 => (map_app_list1 pf l m)
        | (M1 a' y') => (map_app_list1 pf l (MapRemove A m a'))
        | (M2 m'1 m'2) =>
          (list_of_MapDomRestrByDom1 [a0:ad](pf (ad_double_plus_un a0))
            (list_of_MapDomRestrByDom1 [a0:ad](pf (ad_double a0)) l m1 m'1) m2 m'2)
      end
  end.

```

The additional input `pf` of type `ad->ad` is used to define functions on maps using recursion. It has been used at many places in the map library. It is a way of indicating the path from the root of the map to the node of the map which is passed on at a particular recursion depth. Notice the way this function is modified when making a recursive call.

If `m` is the empty map, then the original list is returned. If `m` has only one node in its domain, then it is added to the list depending on whether or not it is absent in the domain of `m'`. If `m` is map with children `m1` and `m2` then we look at the structure of `m'`. If it is empty then we add all the addresses in `m` to the list. For this we use a function `map_app_list1` defined below. If `m'` is a singleton then we add all the addresses in `m` to the list except for the node in `m'`. If `m'` is a map with children `m'1` and `m'2` then we first recursively call the function with the maps `m1` and `m'1`. With the list that is returned from this call we make another recursive call with the maps `m2` and `m'2`. If `m` is map with children `m1` and `m2` then we look at the structure of `m'`. If it is empty then we add all the addresses in `m` to the list. For this we use a function `map_app_list1` defined below. If `m'` is a singleton then we add all the addresses in `m` to the list except for the node in `m'`. If `m'` is a map with children `m'1` and `m'2` then we first recursively call the function with the maps `m1` and `m'1`. With the list that is returned from this call we make another recursive call with the maps `m2` and `m'2`. If `m` is map with children `m1` and `m2` then we look at the structure of `m'`. If it is empty then we add all the addresses in `m` to the list. For this we use a function `map_app_list1` defined below. If `m'` is a singleton then we add all the addresses in `m` to the list except for the node in `m'`. If `m'` is a map with children `m'1` and `m'2` then we first recursively call the function with the maps `m1` and `m'1`. With the list that is returned from this call we make another recursive call with the maps `m2` and `m'2`.

The function `map_app_list1` takes in a list of addresses as input and appends to it all the addresses in the domain of a input map.

### Definition 20

```

Fixpoint map_app_list1 [pf:ad->ad; l:(list ad); m:(Map A)] : (list ad) :=
  Cases m of
    M0 => l
    | (M1 a y) => (cons (pf a) l)
    | (M2 m1 m2) => (map_app_list1 [a0:ad] (pf (ad_double_plus_un a0))
      (map_app_list1 [a0:ad] (pf (ad_double a0)) l m1) m2)
  end

```

end.

For `map_app_list1` we need to prove that the output list contains exactly the nodes which are either in the input list or in the domain of the input map. Also we need to prove that the output list does not have duplicate occurrences of nodes. The assumption is that the input list does not contain any duplicate occurrences. This result is required because we require the free list to be free of duplicate occurrences.

**Lemma 12** *The nodes in the input list are in the output list of `map_app_list1`.*

```
Lemma map_app_list1_lemma_1 :
  (m:(Map A); pf:ad->ad; l:(list ad); a:ad)
  (In a l) -> (In a (map_app_list1 pf l m)).
```

**Proof:** Straightforward using induction on m. □

**Lemma 13** *The nodes in the output list of `map_app_list1` are either in the input list or are in the domain of the input map.*

```
Lemma map_app_list1_lemma_2 :
  (m:(Map A); pf,fp:ad->ad; l:(list ad))
  ((a0:ad)(fp (pf a0))=a0)
  -> (a:ad)(In a (map_app_list1 pf l m))
  -> (In a l)\/(in_dom ? (fp a) m)=true
      /\ (pf (fp a))=a.
```

**Proof:** Using induction on m. □

**Lemma 14** *The output list of `map_app_list1` does not contain any duplicate occurrences if the input list does not contain any duplicate occurrences.*

```
Lemma map_app_list1_lemma_3 :
  (m:(Map A); pf,fp:ad->ad; l:(list ad))
  ((a0:ad)(fp (pf a0))=a0)
  -> (no_dup_list ? l)
  -> ((a:ad)(in_dom ? a m)=true -> ~(In (pf a) l))
  -> (no_dup_list ? (map_app_list1 pf l m)).
```

**Proof:** Using induction on m. □

**Lemma 15** *All nodes in the domain of the input map are also present in the output list.*

```
Lemma map_app_list1_lemma_4 :
  (m:(Map A); pf:ad->ad; l:(list ad))
  (a:ad)(in_dom ? a m)=true
  -> (In (pf a) (map_app_list1 pf l m)).
```

**Proof:** Using induction on  $m$ . □

Next we prove four similar lemmas for the function `list_of_MapDomRestrByDom1`.

**Lemma 16** *All nodes in the input list of `list_of_MapDomRestrByDom1` are also in the output list.*

```
Lemma list_of_MapDomRestrByDom1_lemma_1 :
  (m:(Map A); m':(Map B); l:(list ad); pf:ad->ad)
  (a:ad)(In a l) -> (In a (list_of_MapDomRestrByDom1 pf l m m')).
```

**Proof:** Straightforward using induction on  $m$ . We require the corresponding lemma `map_app_list1_lemma_1`. □

**Lemma 17** *All nodes in the domain of  $m$  and not in the domain of  $m'$  are in the output list of `list_of_MapDomRestrByDom1`.*

```
Lemma list_of_MapDomRestrByDom1_lemma_2:
  (m:(Map A); m':(Map B); l:(list ad); pf:ad->ad)
  (a:ad)(in_dom ? a m)=true
  -> (in_dom ? a m')=false
  -> (In (pf a) (list_of_MapDomRestrByDom1 pf l m m')).
```

**Proof:** Using induction on  $m$ . `map_app_list1_lemma_4` is used. □

**Lemma 18** *All nodes in the output list of `list_of_MapDomRestrByDom1` are either in the input list or are in the domain of  $m$  but not in the domain of  $m'$ .*

```
Lemma list_of_MapDomRestrByDom1_lemma_3:
  (m:(Map A); m':(Map B); l:(list ad); pf,fp:ad->ad)
  ( (a:ad)(fp (pf a))=a )
  -> (a:ad)(In a (list_of_MapDomRestrByDom1 pf l m m'))
  -> (In a l)
  /\ (in_dom ? (fp a) m)=true
  /\ (in_dom ? (fp a) m')=false
  /\ (pf (fp a))=a.
```

**Proof:** Using induction on  $m$ . □

**Lemma 19** *The output list of `list_of_MapDomRestrByDom1` contains no duplicate occurrences provided the input list does not contain any duplicate occurrences.*

```
Lemma list_of_MapDomRestrByDom1_lemma_4 :
  (m:(Map A); m':(Map B); l:(list ad); pf,fp:ad->ad)
  ( (a:ad)(fp (pf a))=a )
  -> ( (a:ad)(in_dom ? a m)=true -> (in_dom ? a m')=false -> ~(In (pf a) l) )
  -> (no_dup_list ? l)
  -> (no_dup_list ? (list_of_MapDomRestrByDom1 pf l m m')).
```

**Proof:** Using induction on  $m$ . □

The new set of free nodes can now be easily computed using `list_of_MapDomRestrByDom1`.

### Definition 21

```
Definition new_fl := [bs:BDDstate; used:(list ad); fl:BDDfree_list]
  (list_of_MapDomRestrByDom1 ? ? [a0:ad]a0 fl bs (mark bs used)).
```

The only result required to be proven about the new free list is that it satisfies all the invariants.

**Lemma 20** *The new list of free nodes is OK.*

```
Lemma new_fl_OK : (bs:BDDstate; used:(list ad); fl:BDDfree_list; counter:ad)
  (BDDstate_OK bs)
  -> (BDDfree_list_OK bs fl counter)
  -> (counter_OK bs counter)
  -> (BDDfree_list_OK (new_bs bs used) (new_fl bs used fl) counter).
```

**Proof:** The absence of duplicate occurrences of nodes in the new list follows from the corresponding result for `list_of_MapDomRestrByDom1`. Now we assume that there is some node `node` in the new free list. We need to show that `node` is between `two` and `counter` and, and that it is not in the new `BDDstate`. In case the node was in the old free list, it was clearly between `two` and `counter` and it was not in the old `BDDstate`, so it is also not in the new `BDDstate` (`new_bs_lemma_1` and `new_bs_lemma_2`). The other case is that the node was in the old `BDDstate` but not in the set of marked nodes. Clearly `node` was greater than `two` because the leaf nodes are not in the domain of `BDDstate`. Also it was less than `counter` because of the conditions in `counter_OK`. Also it is not in the new `BDDstate` which contains only the marked nodes.

Next we assume that `node` is between `two` and `counter` and not in the new `BDDstate`. We need to show that it is in the new free list. The first possibility is that it was not in the old `BDDstate` in which case it was in the old free list and hence also in the new free list. In case it was in the old `BDDstate`, it must not be a marked node since it is not in the new `BDDstate`. Hence it must be in the new free list. □

## 6.4 Cleaning of Sharing and Memoization Maps

Once the unused nodes have been removed from the `BDDstate` and added to the free list, the sharing and memoization maps become inconsistent because they contain references to nodes that have been freed. So we now need to remove all those entries from these maps which refer to such nodes.

The following is the definition of nodes whose references are valid in the sharing and memoization maps. This definition is in terms of any map representing some set of addresses, however for our purposes it would be the set of marked nodes.

**Definition 22** *A reference to a node in the sharing or memoization maps is valid if it is marked node or is a leaf node.*

```

Definition used_node_bs' := [marked:(Map unit); node:ad]
  Cases (MapGet ? marked node) of
    (SOME _) => true
  | NONE => (orb (ad_eq node BDDzero) (ad_eq node BDDone))
end.

```

This definition is required because the way the function mark has been implemented, it does not include the leaf nodes.

The memoization map for negation maps addresses to addresses (which should correspond to their negation). The following function cleans it of invalid references.

### Definition 23

```

Fixpoint clean'1_1 [pf:ad->ad; m':(Map unit); m:(Map ad)] : (Map ad) :=
  Cases m of
    M0 => m
  | (M1 a a') => (if (andb (used_node_bs' m' (pf a))
                        (used_node_bs' m' a')) then m else (M0 ?))
  | (M2 m1 m2) => (makeM2 ? (clean'1_1 [a0:ad](pf (ad_double a0)) m' m1)
                        (clean'1_1 [a0:ad](pf (ad_double_plus_un a0)) m' m2))
end.

```

```

Definition clean'1 := [m:(Map ad); m':(Map unit)](clean'1_1 [a:ad]a m' m).

```

An entry mapping the address  $a$  to the address  $a'$  is valid if both of them satisfy the `used_node_bs'` predicate.

We prove the following result about `clean'1` in terms of `MapGet`.

**Lemma 21** *(clean'1 m m') contains exactly those entries of m in which both the nodes are in the domain of m'.*

```

Lemma clean'1_lemma : (m:(Map ad); m':(Map unit); a,a':ad)
  (MapGet ? (clean'1 m m') a)=(SOME ? a')
  <-> (andb (used_node_bs' m' a) (used_node_bs' m' a'))=true
  /\ (MapGet ? m a)=(SOME ? a').

```

**Proof:** Follows immediately from the following corresponding lemma about `clean'1_1`.  $\square$

### Lemma 22

```

Lemma clean'1_1_lemma : (m:(Map ad); m':(Map unit); pf:ad->ad)
  (a,a':ad)(MapGet ? (clean'1_1 pf m' m) a)=(SOME ? a')
  <-> (andb (used_node_bs' m' (pf a)) (used_node_bs' m' a'))=true
  /\ (MapGet ? m a)=(SOME ? a').

```

**Proof:** Straightforward using induction on  $m$ . □

Next we describe the cleaning of the memoization map for disjunction. The memoization map is supposed to represent a function from pairs of addresses to addresses. This is represented using maps by the data type `(Map (Map ad))` which represents maps from addresses to maps from addresses to addresses. The lookup function for such maps can be defined as follows.

### Definition 24

```
Definition MapGet2 := [m:(Map (Map A)); a,b:ad]
  Cases (MapGet ? m a) of
    (NONE) => (NONE A)
  | (SOME m') => (MapGet ? m' b)
end.
```

The following function `clean'2` is used for cleaning of such two level maps. It is defined using the corresponding function for the one level map (`clean'1`).

### Definition 25

```
Fixpoint clean'2_1 [pf:ad->ad; m':(Map unit); m:(Map (Map ad))]
  : (Map (Map ad)) :=
  Cases m of
    M0 => m
  | (M1 a y) => (if (used_node_bs' m' (pf a))
    then (M1 ? a (clean'1 y m')) else (M0 ?))
  | (M2 m1 m2) => (makeM2 ? (clean'2_1 [a0:ad](pf (ad_double a0)) m' m1)
    (clean'2_1 [a0:ad](pf (ad_double_plus_un a0)) m' m2))
end.
```

```
Definition clean'2 := [m:(Map (Map ad)); m':(Map unit)](clean'2_1 [a:ad]a m' m).
```

The results proved for them are similar to those for the one level functions. The proofs make use of the results for the one level functions.

### Lemma 23

```
Lemma clean'2_1_lemma : (m:(Map (Map ad)); m':(Map unit); pf:ad->ad)
  (a,b,c:ad)(MapGet2 ? (clean'2_1 pf m' m) a b)=(SOME ? c)
<-> (andb (used_node_bs' m' (pf a))
  (andb (used_node_bs' m' b) (used_node_bs' m' c)))=true
/\ (MapGet2 ? m a b)=(SOME ? c).
```

```
Lemma clean'2_lemma : (m:(Map (Map ad)); m':(Map unit))
```

```

(a,b,c:ad)(MapGet2 ? (clean'2 m m') a b)=(SOME ? c)
<-> (andb (used_node_bs' m' a)
      (andb (used_node_bs' m' b) (used_node_bs' m' c)))=true
/\ (MapGet2 ? m a b)=(SOME ? c).

```

Next we need to prove that these function do what we require for the memoization maps, i.e. the new maps after garbage collection are consistent with the rest of the configuration.

The following is the definition of the conditions when a memoization table for negation is OK. The definition for disjunction is similar.

```

Definition BDDneg_memo_OK := [bs:BDDstate; negm:BDDneg_memo]
  (node,node':ad)
  (MapGet ? negm node)=(SOME ? node')
-> (node_OK bs node)
  /\ (node_OK bs node')
  /\ (ad_eq (bs_var' bs node') (bs_var' bs node))=true
  /\ (bool_fun_eq (bool_fun_of_BDD_bs bs node')
                (bool_fun_neg (bool_fun_of_BDD_bs bs node))).

```

A memoization table for negation is OK if whenever it maps `node` to `node'`, then both `node` and `node'` are OK, the variables stored at the two nodes are equal and the boolean function represented by `node` is negation of that represented by `node'`.

**Lemma 24** *The new memoization table for negation returned by `clean'1` is consistent with the new BDDstate.*

```

Lemma new_negm_OK : (bs:BDDstate; used:(list ad); negm:BDDneg_memo)
  (BDDstate_OK bs)
-> (BDDneg_memo_OK bs negm)
-> (BDDneg_memo_OK (new_bs bs used) (clean'1 negm (mark bs used))).

```

**Proof:** The fact that the nodes are OK follows from the fact that all the marked nodes are present in the `newBDDstate`. The variables at the two nodes were equal in the old `BDDstate` and they are preserved in the new `BDDstate`. We need to show that the boolean functions represented by the nodes are also preserved in the new `BDDstate`. This is stated below.  $\square$

**Lemma 25**

```

Lemma node_preserved_bs_bool_fun :
  (bs,bs':BDDstate; node:ad)
  (BDDstate_OK bs)
-> (BDDstate_OK bs')
-> (node_preserved_bs bs bs' node)
-> (node_OK bs node)
-> (bool_fun_eq (bool_fun_of_BDD_bs bs' node) (bool_fun_of_BDD_bs bs node)).

```

**Proof:** Follows immediately from the following lemma. □

### Lemma 26

```

Lemma node_preserved_bs_bool_fun_1 :
  (n:nat; bs,bs':BDDstate; node:ad)
  (BDDstate_OK bs)
  -> (BDDstate_OK bs')
  -> (node_preserved_bs bs bs' node)
  -> (node_OK bs node)
  -> n=(nat_of_ad (bs_var' bs node))
  -> (bool_fun_eq (bool_fun_of_BDD_bs bs' node) (bool_fun_of_BDD_bs bs node)).

```

**Proof:** An additional variable  $n$  has been introduced to enable us to apply induction. The proof is by well founded induction on  $n$ . It is straightforward but a bit long. □

The above lemmas about boolean functions are stated in terms of the term `node_preserved_bs` whereas the lemmas about the functions `clean'1` and `clean'2` were stated in terms of `used_node_bs'`. We have the following lemma to relates the two notions.

### Lemma 27

```

Lemma used_node_bs'_preserved :
  (bs:BDDstate; used:(list ad); node:ad)
  (BDDstate_OK bs)
  -> (used_node_bs' (mark bs used) node)=true
  -> (node_preserved_bs bs (new_bs bs used) node).

```

Next we describe the cleaning of the sharing maps. The definition of the sharing maps is similar to that of the memoization maps. The sharing map needs to map triples of a variable and two addresses to addresses. Since variables have also been represented by the same data type as addresses it is possible to define the sharing maps as three level maps.

We give here the definitions of the functions used for the cleaning of the sharing map. These are very similar to the ones for the memoization maps. The only difference is that one of the components in the tuple is a variable instead of a node so we do not need to impose any condition it. The lemmas proved about these functions are also similar so we have not stated it here, instead we have just given the final result for the sharing map.

We remark that while we have written separate functions for cleaning of the various sharing and memoization maps, it should be possible to define a general  $n$  level map and write a function for removing unwanted nodes from it for some general condition on the nodes which are to be retained.

```

Fixpoint clean1 [m':(Map unit); m:(Map ad)] : (Map ad) :=

```

```

Cases m of
  M0 => m
  | (M1 a a') => (if (used_node_bs' m' a') then m else (M0 ?))
  | (M2 m1 m2) => (makeM2 ? (clean1 m' m1) (clean1 m' m2))
end.

```

```

Fixpoint clean2_1 [pf:ad->ad; m':(Map unit); m:(Map (Map ad))]
: (Map (Map ad)) :=

```

```

Cases m of
  M0 => m
  | (M1 a y) => (if (used_node_bs' m' (pf a)) then (M1 ? a (clean1 m' y))
                else (M0 ?))
  | (M2 m1 m2) => (makeM2 ? (clean2_1 [a0:ad](pf (ad_double a0)) m' m1)
                          (clean2_1 [a0:ad](pf (ad_double_plus_un a0)) m' m2))
end.

```

```

Definition clean2 := [m:(Map (Map ad)); m':(Map unit)](clean2_1 [a:ad]a m' m).

```

```

Fixpoint clean3_1 [pf:ad->ad; m':(Map unit); m:(Map (Map (Map ad)))]
: (Map (Map (Map ad))) :=

```

```

Cases m of
  M0 => m
  | (M1 a y) => (if (used_node_bs' m' (pf a)) then (M1 ? a (clean2 y m'))
                else (M0 ?))
  | (M2 m1 m2) => (makeM2 ? (clean3_1 [a0:ad](pf (ad_double a0)) m' m1)
                          (clean3_1 [a0:ad](pf (ad_double_plus_un a0)) m' m2))
end.

```

```

Definition clean3 := [m:(Map (Map (Map ad))); m':(Map unit)]
(clean3_1 [a:ad]a m' m).

```

```

Fixpoint clean3_1 [pf:ad->ad; m':(Map unit); m:(Map (Map (Map ad)))]
: (Map (Map (Map ad))) :=

```

```

Cases m of
  M0 => m
  | (M1 a y) => (if (used_node_bs' m' (pf a)) then (M1 ? a (clean2 y m'))
                else (M0 ?))
  | (M2 m1 m2) => (makeM2 ? (clean3_1 [a0:ad](pf (ad_double a0)) m' m1)
                          (clean3_1 [a0:ad](pf (ad_double_plus_un a0)) m' m2))
end.

```

```

Definition clean3 := [m:(Map (Map (Map ad))); m':(Map unit)]
(clean3_1 [a:ad]a m' m).

```

**Lemma 28** *The new sharing map returned by clean3 is OK.*

```

Lemma new_share_OK : (bs:BDDstate; used:(list ad); share:BDDsharing_map)
  (BDDstate_OK bs) -> (BDDsharing_OK bs share)
  -> (BDDsharing_OK (new_bs bs used) (clean3 share (mark bs used))).

```

## 6.5 Combining Everything

In the new configuration we keep the counter the same as in the old configuration. The invariants on the counter are the same as in the previous implementation, that it should be greater than two and there should be no node stored after it.

```

Definition counter_OK := [bs:BDDstate; counter:ad]
  (ad_le (ad_x (x0 xH)) counter)=true
  /\ (a:ad)(ad_le counter a)=true
  -> (MapGet ? bs a)=(NONE ?).

```

**Lemma 29** *The counter in the new configuration is OK.*

```

Lemma new_counter_OK : (bs:BDDstate; used:(list ad); counter:ad)
  (BDDstate_OK bs) -> (counter_OK bs counter)
  -> (counter_OK (new_bs bs used) counter).

```

**Proof:** The first condition in `counter_OK`, viz that counter is greater than two clearly holds since the counter was OK in the previous configuration. Also from `new_bs_lemma_1` and `new_bs_lemma_2`, and the fact that there was nothing stored after counter in the previous implementation, it follows that there is nothing stored after the counter in the new configuration.  $\square$

Combining the previous functions, we get the garbage collector for the configuration.

### Definition 26

```

Inductive dummy_mark : Set := DM : (Map unit) -> dummy_mark.

```

```

Definition gc_0 := [cfg:BDDconfig; used:(list ad)]
  Cases cfg of (bs,(share,(fl,(counter,(negm,orm)))) =>
    Cases (DM (mark bs used)) of (DM marked) =>
      let bs' = (MapDomRestrTo ? ? bs marked) in
      let fl' = (list_of_MapDomRestrByDom1 ? ? [a0:ad]a0 fl bs marked) in
      let share' = (clean3 share marked) in
      let negm' = (clean'1 negm marked) in
      let orm' = (clean'2 orm marked) in
      (bs',(share',(fl',(counter,(negm',orm')))))
    end
  end.

```

This function can be shown to satisfy the requirements for a garbage collector that were mentioned in the beginning.

**Lemma 30**  $gc_0$  satisfies the requirements for a garbage collector.

Lemma  $gc\_0\_OK$  :  $(gc\_OK\ gc\_0)$ .

**Proof:** The proof follows by straightforward applications of the results for the individual functions. □

# Chapter 7

## Integration of the Garbage Collector into the BDD algorithms

The next step after implementing the garbage collection routines is to integrate it into the rest of the BDD algorithms. This is to enable the calling of the garbage collector automatically from within the algorithms whenever the need arises. In this chapter we describe how we changed these functions to allow for garbage collection. These functions take an arbitrary garbage collector as input, which satisfies the requirements stated in `gc_OK`. This allows us to change the garbage collection policy whenever we like, without changing the rest of the implementation. We give here an example of how we can change this policy.

We have the following trivial example which does not do any garbage collection, i.e. it returns the same configuration.

### Definition 27

```
Definition gc_inf := [cfg:BDDconfig; used:(list ad)]cfg.
```

This function also satisfies all the requirements stated in `gc_OK`.

### Lemma 31

```
Lemma gc_inf_OK : (gc_OK gc_inf).
```

Now given any condition `cond` of type `BDDconfig -> bool`, we could define a new garbage collector which would return `(gc_0 cfg)` if `(cond cfg)` is true, otherwise it would return `(gc_inf cfg)`. This can easily be shown to satisfy the conditions in `gc_OK`.

This condition could take into account, for example, the number of nodes allocated, whether there are free nodes or not, etc. We could also decide to keep some more information with the configuration which would help us in deciding whether to call the garbage collector or not. This new approach should also be simple enough to prove.

## 7.1 Allocation

The first step in modifying the rest of the BDD algorithms is to modify the function which adds new nodes into the configuration. All the logical operations on BDDs make use of a function `BDDmake` which takes in as input a variable and two nodes in the configuration. If the two nodes are same, it returns that as a result. Otherwise it returns the node which contains that variable and the two nodes as children. If such a node was already present in the configuration (found by looking up the sharing map), then that node is returned, otherwise a new node needs to be added into the configuration.

We first describe here the function which adds a new node into the configuration. It is common to call the garbage collector from within the allocation function itself, depending on some policy.

In the previous implementation, the new node was simply added at the counter address, and the counter was incremented. A corresponding entry was made into the sharing map.

In the new implementation, the allocator takes as an additional input a list of used nodes representing the BDDs required by the user. Also it takes the garbage collector as input, so that we can change the garbage collection policy whenever we like. The allocator first calls the garbage collector. Then it checks whether there is some free address in the free list. If so, it takes the one at the beginning of the list, then allocates the node at that address. If there are no free nodes, the old allocation policy is used.

### Definition 28

```

Definition BDDalloc := [gc:(BDDconfig->(list ad)->BDDconfig);
                        cfg:BDDconfig; x:BDDvar; l,r:ad; ul:(list ad)]
Cases new_cfg of
  (bs,(share,(fl,(counter,(negm,orm)))) =>
    Cases fl of
      (cons a fl') => (((MapPut ? bs a (x,(l,r))),
                       ((MapPut3 ? share l r x a),
                        (fl',(counter,(negm,orm))))),a)
    | nil => (((MapPut ? bs counter (x,(l,r))),
              ((MapPut3 ? share l r x counter),
               (fl,((ad_S counter),(negm,orm))))),counter)
    end
  end.

```

We now discuss the hypotheses on the arguments. Besides the conditions in the previous implementation, we require that the input nodes `l` and `r` should be used nodes, i.e. they should be reachable from the nodes in the used list (or they should be leaf nodes). This is necessary because otherwise they could be removed by the garbage collector. In the previous implementation we just required them to be present in the `BDDstate`.

```

Hypothesis l_used' : (used_node' cfg ul l).
Hypothesis r_used' : (used_node' cfg ul r).

```

The predicate `used_node'` is defined below.

```
Definition used_node_bs := [bs:BDDstate; ul:(list ad); node:ad]
  (EX node':ad | (In node' ul) /\ (nodes_reachable bs node' node)).
```

```
Definition used_node'_bs := [bs:BDDstate; ul:(list ad); node:ad]
  node=BDDzero \/ node=BBDone \/ (used_node_bs bs ul node).
```

```
Definition used_node := [cfg:BDDconfig](used_node_bs (Fst cfg)).
```

```
Definition used_node' := [cfg:BDDconfig](used_node'_bs (Fst cfg)).
```

Finally we have the condition that the garbage collector passed as input is OK.

```
Hypothesis gc_is_OK : (gc_OK gc).
```

As mentioned in section 6.1 one of the requirements of the garbage collector is that it does not add any nodes into the configuration which were not there in the original configuration. This is required because the allocator is called by `BDDmake` only when the required node is not already present in the configuration. If the garbage collector arbitrarily adds a node into the configuration, it could be the same as the one required to be allocated (i.e. contain the same variable and point to the same children). This could lead to duplication of nodes in the configuration, violating the invariants defined on the configuration.

Next we define the `BDDmake` function which calls the `BDDalloc` function.

### Definition 29

```
Definition BDDmake := [gc:(BDDconfig->(list ad)->BDDconfig);
  cfg:BDDconfig; x:BDDvar; l,r:ad; ul:(list ad)]
if (ad_eq l r) then (cfg,l) else
  Cases (MapGet3 ? (Fst (Snd cfg)) l r x) of
    (SOME y) => (cfg,y)
  | NONE => (BDDalloc gc cfg x l r ul)
end.
```

The results proved for the allocation and `BDDmake` function are similar to the ones in the previous implementation, namely, that the output configuration and node are OK, the node has the required semantics in terms of boolean functions. However in the previous implementation, we proved that all the nodes from the old configuration are preserved in the new configuration. This no longer holds because the garbage collector can be called in between. So we instead have the following result that the used nodes are preserved.

### Lemma 32

```
Lemma BDDmake_preserves_used_nodes : (gc:(BDDconfig->(list ad)->BDDconfig))
```

```

(gc_OK gc)
->(cfg:BDDconfig; x:BDDvar; l,r:ad; ul:(list ad))
(BDDconfig_OK cfg)
->(used_list_OK cfg ul)
->(used_nodes_preserved cfg (Fst (BDDmake gc cfg x l r ul)) ul).

```

## 7.2 Logical Operations

The main logical operations as before are negation and disjunction. The rest (conjunction, implication, double implication) are implemented in terms of these two.

As with the allocation function, these operations take in a garbage collector as a parameter. They also take in as input the list of used nodes required by the user. The functions work as before except that now the allocator function has changed, so the garbage collector can be called in the middle of execution. So these functions must take care to add the results of the intermediate computation to the list of used nodes, so that they are not lost. They can be removed later once they are not required. We give here the new definition of the negation function as an example. The changes made in the other operations are also similar.

### Definition 30

```

Fixpoint BDDneg_1 [cfg:BDDconfig; ul:(list ad); node:ad; bound:nat]
  : BDDconfig*ad :=
Cases bound of
  0 => (* Error *) (initBDDconfig,BDDzero)
| (S bound') =>
Cases (MapGet ? (negm_of_cfg cfg) node) of
  (SOME node') => (cfg,node')
| NONE =>
Cases (MapGet ? (Fst cfg) node) of
  NONE => (if (ad_eq node BDDzero) then
    ((BDDneg_memo_put cfg BDDzero BDDone),BDDone) else
    ((BDDneg_memo_put cfg BDDone BDDzero),BDDzero))
| (SOME (x,(l,r))) =>
Cases (BDDneg_1 cfg ul l bound') of (cfgl,nodel) =>
  Cases (BDDneg_1 cfgl (cons nodel ul) r bound') of (cfgr,noder) =>
    Cases (BDDmake gc cfgr x nodel noder (cons noder (cons nodel ul))) of
      (cfg',node') => ((BDDneg_memo_put cfg' node node'),node')
    end
  end
end
end
end
end
end.

```

Once the negation of the left node  $l$  has been calculated, it is added to the used list  $ul$  before computing the negation  $r$ . Finally after `BDDmake`, these nodes are no longer required because these can any way be reached from the output node.

We prove the following result about the negation function.

### Lemma 33

Lemma `BDDneg_1_lemma` :

```
(bound:nat; cfg:BDDconfig; ul:(list ad); node:ad)
(lt (nat_of_ad (var' cfg node)) bound)
-> (BDDconfig_OK cfg)
-> (used_list_OK cfg ul)
-> (used_node' cfg ul node)
-> (BDDconfig_OK (Fst (BDDneg_1 cfg ul node bound)))
  /\ (config_node_OK (Fst (BDDneg_1 cfg ul node bound))
      (Snd (BDDneg_1 cfg ul node bound)))
  /\ (used_nodes_preserved cfg (Fst (BDDneg_1 cfg ul node bound)) ul)
  /\ (ad_eq (var' (Fst (BDDneg_1 cfg ul node bound))
              (Snd (BDDneg_1 cfg ul node bound)))
      (var' cfg node))=true
  /\ (bool_fun_eq (bool_fun_of_BDD (Fst (BDDneg_1 cfg ul node bound))
                                  (Snd (BDDneg_1 cfg ul node bound)))
                 (bool_fun_neg (bool_fun_of_BDD cfg node)))).
```

We prove the following similar lemma about the disjunction function.

### Lemma 34

Lemma `BDDor_1_lemma` :

```
(bound:nat; cfg:BDDconfig; ul:(list ad); node1,node2:ad)
(lt (max (nat_of_ad (var' cfg node1)) (nat_of_ad (var' cfg node2))) bound)
-> (BDDconfig_OK cfg)
-> (used_list_OK cfg ul)
-> (used_node' cfg ul node1)
-> (used_node' cfg ul node2)
-> (BDDconfig_OK (Fst (BDDor_1 cfg ul node1 node2 bound)))
  /\ (config_node_OK (Fst (BDDor_1 cfg ul node1 node2 bound))
      (Snd (BDDor_1 cfg ul node1 node2 bound)))
  /\ (used_nodes_preserved cfg (Fst (BDDor_1 cfg ul node1 node2 bound)) ul)
  /\ (ad_le (var' (Fst (BDDor_1 cfg ul node1 node2 bound))
              (Snd (BDDor_1 cfg ul node1 node2 bound)))
      (BDDvar_max (var' cfg node1) (var' cfg node2)))=true
  /\ (bool_fun_eq (bool_fun_of_BDD (Fst (BDDor_1 cfg ul node1 node2 bound))
                                  (Snd (BDDor_1 cfg ul node1 node2 bound)))
                 (bool_fun_or (bool_fun_of_BDD cfg node1)
                              (bool_fun_of_BDD cfg node2)))).
```

**Proof:** The statement of the lemma is similar to that in the previous implementation, with some changes. For example, we have the extra condition that the input used list is OK. Also we have the condition that `node1` and `node2` are used nodes, instead of being just present in the `BDDstate`. We have the usual condition that the input configuration is OK (which includes the condition that the memoization maps are OK). In the conclusions, besides having the required property of `BDDor_1` in terms of boolean functions, we also have the conditions that the configuration returned is OK, the node returned is OK, the used nodes of the input configuration are preserved in the output configuration and the variable stored at the output node is less than or equal to the variables stored at the input nodes.

The proof is by induction on bound. The structure of the proof is similar to that without garbage collection. We require to prove all the conclusions together and they cannot be proved as separate lemmas because they are required in the induction hypotheses. For example, the function `first` recursively calls the disjunction of the left nodes. This returns a new configuration in which the disjunction of the right nodes is recursively called. For this we require that the configuration returned after the first recursive call should be OK. Also we require that the meaning of the right nodes in that configuration is same as what they were in the original configuration. Besides, of course, we require that the node returned after the first recursive call corresponds to the disjunction of the left nodes.

The main lemmas used are those related to `BDDmake`. We use the semantics of `BDDmake` in terms of boolean functions besides the fact that the configuration and node returned by it are OK. The condition on the variables mentioned in the statement of the lemma is required because `BDDmake` requires the variable at its input node to be less than the input variable.

The main difference from the proof without disjunction is because of the fact that we need to take care of the list of used nodes. For example, after the disjunction of the left nodes has been computed, it is added to the list of the used nodes before calling the disjunction of the right nodes, because it is later used by `BDDmake`. This ensures that it is not removed by the garbage collector.  $\square$

The proofs for disjunction and negation are the largest proofs in the implementation (as they were in the previous implementation) although they have been considerably simplified now. The main problem with them is that we have to worry more about the configuration rather than the actual BDDs on which the operations have to be performed. We have to frame complex induction hypotheses like the output node and configuration are OK, the used nodes are preserved, etc, besides the required property of the operations in terms of boolean functions. Also it is difficult to prove these results separately and have to be proven together.

The rest of the logical operations are implemented in terms of these two. The changes made to them are similar. We finally have the tautology checker `is_tauto` which works by building the BDD for a given boolean expression and checking whether the final BDD is the leaf node `BDDone`. It works for an arbitrary garbage collector as input. We have the following correctness result for the tautology checker.

### Lemma 35

Lemma `is_tauto_lemma` : `(gc:(BDDconfig->(list ad)->BDDconfig))`

```
(gc_OK gc)
->(be:bool_expr)
(is_tauto gc be)=true
<->(bool_fun_eq bool_fun_one (bool_fun_of_bool_expr be))
```

# Chapter 8

## Experiments

We conducted a few simple experiments on one formula examples to measure their sizes. We dealt with the following three kinds of formulas.

- Urquhart's U-formulas.  $U_n = x_1 \Leftrightarrow (x_2 \Leftrightarrow \dots \Leftrightarrow (x_n \Leftrightarrow (x_1 \Leftrightarrow (x_2 \Leftrightarrow \dots \Leftrightarrow x_n))))$ . These are tautologies.
- Pigeon hole formulas. They code the fact that we cannot put  $n + 1$  pigeons in  $n$  holes
- N Queen's problem. We coded the formula that represents the set of possible arrangements of  $n$  queens on a  $n \times n$  chess board such that no queen attacks any other queen. These again are tautologies.

Out of these the first two (besides others) had been used for extensive benchmarking of the previous implementation [VGL00] to measure the time and space requirements of the implementation. We mainly present here the sizes of the BDDs produced in the presence of the garbage collector and compare with the total number of nodes that need to be allocated otherwise.

The most dramatic improvement in the space requirements was seen in case on the Urquhart's formulas.

n	total nodes required without gc
10	570
20	2140
30	4710
40	8280
50	12850

Figure 8.1: Number of nodes required for  $U_n$  without garbage collections

n	total nodes required without gc	minimum nodes required with gc
1	8	
2	70	
3	270	
4	854	
5	2498	2000
6	7006	5000
7	19030	14000
8		36000

Figure 8.2: pigeon hole formulas

We require to allocate 12850 nodes for  $U_{50}$  in absence of garbage collection. However in presence of garbage collection, we could build BDDs for formulas upto  $U_{100}$  without allocating more than 2000 nodes. This means that the intermediate BDDs involved in building up the BDD for the whole formula never grow bigger than 2000 nodes (the final BDD of course contains only one node since these formulas are tautologies).

These figures were obtained by fixing some bound on the number of nodes to be allocated and calling the garbage collector whenever the counter is equal to that value and no node is free.

While in the above example, use of garbage collection increased the size of problems that could be solved using them, this was not found to be so in case of the other two kinds of formulas. This is because the maximum size of the intermediate BDDs required during the computation are of sizes comparable to to the total number of nodes allocated in absence of garbage collection.

Still we can see that once the BDD has been built, the final BDD produced is much smaller in size. This can be seen in Figure 8.3 where for  $n = 5$  (six queens) the final BDD has only 131 nodes where as the total nodes required without garbage collection is 64554. The minimum nodes required with garbage collection is 34000. In the other two examples of course the final BDDs are singleton nodes. This shows an obvious advantage of calling the garbage collector once the BDD has been built, in case we require to build BDDs for more formulas.

The figures for the minimum nodes required have been measured in units of thousand nodes and should therefore not be regarded as exact values. For example, the figure of 34000 nodes in Figure 8.3 indicates that at least 33000 nodes are required but not more than 34000 nodes are required.

n	total nodes required without gc	minimum nodes required with gc	size of final BDDs
0	4		3
1	58		1
2	444		1
3	2068	1000	31
4	10172	5000	69
5	64554	34000	131

Figure 8.3: N Queens formula

# Chapter 9

## Conclusion

We managed to implement and formally prove the correctness of a mark and sweep garbage collector for the BDD implementation in Coq. The existing implementation was modified to work in presence of a garbage collector. The whole thing together was then formally proved to work correctly. Some experiments conducted on single formulas show the vast difference in the sizes of final BDDs for various formulas and the total nodes required during the computation of those BDDs. This implementation paves the way for writing a symbolic model checker in Coq using the basic BDD algorithms.

# Bibliography

- [BBC<sup>+</sup>99] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Sabi, and Benjamin Werner. The Coq proof assistant reference manual. Version 6.2.41, available at <http://coq.inria.fr/doc/main.html>, January 1999.
- [GL98] Jean Goubault-Larrecq. Satisfiability of inequality constraints and detection of cycles with negative weight in graphs. Part of the Coq contribs, available at <http://pauillac.inria.fr/coq/contribs/graphs.html>, 1998.
- [HKPM98] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq Proof Assistant, A Tutorial*. Coq Project, Inria, 1998. Draft, version 6.2.4. Available at <http://coq.inria.fr/doc/tutorial.html>.
- [Ver99] Kumar Neeraj Verma. Bdd algorithms and proofs in coq, by reflection. Part of the Coq contribs, available at <http://pauillac.inria.fr/coq/contribs/bdds.html>, 1999.
- [VGL00] Kumar Neeraj Verma and Jean Goubault-Larrecq. Reflecting bdds in coq. Research Report RR-3859, INRIA, 2000.