HAL
archives-ouvertes.fr

Reflecting BDDs in Coq

Kumar Neeraj Verma, Jean Goubault-Larrecq

▶ **To cite this version:**

Kumar Neeraj Verma, Jean Goubault-Larrecq. Reflecting BDDs in Coq. [Research Report] RR-3859, INRIA. 2000. <inria-00072797>

HAL Id: inria-00072797
https://hal.inria.fr/inria-00072797

Submitted on 24 May 2006

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Reflecting BDDs in Coq

Kumar Neeraj Verma and Jean Goubault-Larrecq

N ° 3859
Janvier 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--3859--FR+ENG

Reflecting BDDs in Coq

Kumar Neeraj Verma* and Jean Goubault-Larrecq**

Thème 2 — Génie logiciel
et calcul symbolique

Projet Coq

Rapport de recherche n°3859 — Janvier 2000

Abstract: We propose an implementation and a proof of correctness of binary decision diagrams (BDDs), completely formalized in Coq. This not only shows that BDD algorithms are correct, but also yields a certified BDD algorithm running in Caml, using extraction, and even allows us to run BDD-based algorithms inside Coq itself, by reflection. The latter point paves the path for a smooth integration of symbolic model-checking in the Coq proof assistant. Extensive experimental results show that this approach works in practice, and is able to solve both relatively hard propositional problems and actual industrial hardware verification problems.

Key-words: Binary decision diagrams, BDD, Caml, Coq, extraction, proof, reflection

(Résumé : tsvp)

This work has been done in the context of Dyade (R&D joint venture between Bull and Inria).

*csu96139@cse.iitd.ernet.in, on summer leave from IIT Delhi.
**Jean.Goubault@{dyade,inria}.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

La réflexion de diagrammes de décision binaire en Coq

Résumé : Nous proposons une réalisation et une preuve de correction des diagrammes de décision binaires, ou BDD, complètement formalisée en Coq. Ceci montre non seulement que les algorithmes de BDD sont corrects, mais fournit aussi un algorithme certifié de BDD sous Caml, par extraction, et nous permet même de faire tourner les algorithmes de BDD en Coq lui-même, par réflexion. Ce dernier point indique comment intégrer élégamment un model-checker symbolique dans l'assistant à la preuve Coq. Une étude expérimentale est effectuée, qui montre que cette approche fonctionne en pratique, et permet de résoudre tant des problèmes propositionnels relativement durs que de vrais problèmes industriels de vérification de matériel.

Mots-clé : BDD, Caml, Coq, diagrammes de décision binaire, extraction, preuve, réflexion.

Introduction

Binary Decision Diagrams (BDDs for short [Bry86]) are a compact and canonical representation of propositional formulae up to propositional equivalence, or equivalently of Boolean functions. BDDs and related data structures are at the heart of modern automated hardware verification systems, based on model-checking [McM93] or on direct evaluation of observable equivalence between finite-state machines [Cou91]. Not only are these systems automatic, they are also able to solve huge verification problems.

On the other hand, proof assistants like Coq [BBC+99], Lego [LP92] or HOL [GM93] provide expressiveness and safety. They are *expressive* in that not only finite-state machines or Boolean functions can be expressed in such systems, but actually most of mathematics. From the viewpoint of verification, this helps notably in expressing modular properties of circuits [KSK93]. As a toy example, consider a processor with an adder A and a memory subsystem M connected in such a way that the adder takes its inputs in memory through M and writes the results to memory through M again. It is much easier to prove that this does the expected thing by showing that, first, A obeys the general specification of adders; second, that M has all the properties of memory subsystems; and third, that any adder (not just A) connected in the specified way to any memory subsystem (not just M) will run as expected. In so doing, it is useful to abstract away from pesky details like bus width, latency, and so on, and this takes us out of the realm of propositional logic. More importantly, higher-order logic also helps find concise arguments for correctness.

Proof assistants are also *safe*, in that they rest on firm logical foundations, for which consistency proofs are known. (At least, this is the case for HOL and Coq.) Implementations of proof assistants might of course be faulty, but some design principles help limit this to an absolute minimum. In HOL, the main design principle, inherited from LCF [GMW77], is that every deduction, whatever its size may be, must be justified in terms of a a small number of undisputable elementary logical principles. In Coq and Lego, every proof-search phase is followed by an independent proof-checking phase, so that buggy proof-search algorithms cannot fool the system into believing logically incorrect arguments.

But proof assistants are not automatic, even when it comes to specialized domains like Boolean functions. In particular, proof assistants today cannot do model-checking. If the proof assistant is to remain safe, we cannot just force it to rely on the results provided by an external model-checker. We might instrument an external model-checker so that it returns an actual proof, which the given proof assistant will then be able to check. But model-checkers actually enumerate state spaces of astronomical sizes, and mimicking this enumeration—at least naively, see Section 1–with proof rules inside the proof assistant would be foolish.

One remedy to this model-checker (automatic, fast) vs. proof assistant (expressive, safe) antinomy is to use *reflection* [Wey80, BM81, ACHA90, BC93, Bou97]. The general idea to prove a given property P applied to some term t is as follows. Assume we have a proof assistant in which you can also describe and prove programs: NQTHM [BM79], Coq and Lego are three examples. Then write a program π that takes t as input and returns true exactly when $P(t)$ holds. For example, t will denote a propositional formula, and π will compute the BDD of t and compare it to the BDD representing true, whereas $P(t)$ asserts that t is valid. Now show that π is correct with respect to P: whenever $\pi(t)$ returns true, then $P(t)$ holds. In fact, show this inside the proof assistant itself. Now whenever you wish to prove a property of the form $P(u)$, just compute $\pi(u)$, and if this returns true, then apply the correctness lemma to infer $P(u)$: the proof of $P(u)$ has been replaced by the computation of $\pi(u)$.

Reflection is particularly adequate in Coq, since Coq is based on the calculus of inductive constructions, a logic which is essentially a typed lambda-calculus, hence essentially a programming language. Reflection in Coq was pioneered by Samuel Boutin [Bou97], who used it in particular to decide efficiently whether two expressions, denoting values in a ring, are equal. He demonstrated that reflection enabled one to decide equations that are out of reach of standard proof techniques, i.e. not using reflection. Based on this encouraging experiment, it only seems natural to try reflection again, as a mechanism for integrating BDDs in Coq, in a way that would provide the best of both worlds: expressiveness and safety from Coq, automation and speed by using an implementation of BDDs.

Be warned, though, that implementing BDDs in some lambda-calculus will necessarily exact some speed penalty. So one other aim of this work is to evaluate how (un)reasonable it is to do so. Our long-term goal is to produce certified model-checkers, and also reflected model-checkers that could work as subsystems of Coq. Note in particular that we don't just want to use BDDs as validity checkers, as the example above

might have suggested, but as actual representations of sets of states, of Boolean functions and of transition relations [McM93]. This paper presents a first step in this direction, by describing how we managed to fit a library of BDD algorithms inside Coq, and how we proved it correct formally.

The plan of the paper is as follows. We start by reviewing some related works in Section 1, then give short descriptions of BDDs in Section 2, and of Coq in Section 3. As far as the latter is concerned, we shall only describe the features that will be required later on. Section 4 is the technical core of this paper: this is where we describe how BDDs were not only modeled but actually programmed and proved in Coq. This involves a number of difficulties, to name a few: we have to describe memory allocation, sharing, recursion over BDDs, memoization, and to prove that they all work as expected. In Section 5 we discuss experimental results, and discuss speed and space issues. We then conclude in Section 6.

1 Related Works

One piece of work very much related to ours is John Harrison's interfacing of a BDD library with the HOL prover [Har95]. The author's goal is not to do any model-checking, rather to solve the validity problem for propositional logic formulae in an efficient way. Because the logical language of HOL does not contain a programming sublanguage as NQTHM or Coq, reflection is not an option: the BDD library must log every BDD reduction step and translate each as a proper HOL inference, which HOL will then check.

This logging process is difficult to implement, produces huge logs, and it is necessary to use a few tricks in the HOL implementation to keep the HOL proof-checking phase reasonably efficient. It is in fact much easier and much more efficient to use Gunnar Stålmarck's algorithm [Stå94], as J. Harrison shows in a later paper [Har96].

Moreover, even when we do not care about proof-checking, and only long for an efficient validity-checking procedure, BDDs are in general not the best tool: the Davis-Putnam method [DP60, DLL62] and Stålmarck's algorithm are two serious contenders. As far as the former is concerned, there are situations where BDDs are more efficient, and situations where Davis-Putnam wins [US94]. As far as the latter is concerned, Stålmarck's algorithm is able to solve industrial propositional validity problems much larger than what Davis-Putnam or BDDs can handle [Bor97]. But there are still a few cases where BDDs fare better [Bry99]. Moreover, J. Moore has already suggested that the formal verification of BDD algorithms might be tractable [Moo94].

In any case, and although BDDs can be used for deciding propositional validity, they mostly shine in model-checking of temporal logic formulae, which are out of reach of Davis-Putnam's and Stålmarck's algorithms [McM93]. Indeed, we chose to implement and prove BDDs in Coq as a first step towards having a certified and integrated model-checker inside Coq itself. Deciding propositional validity in an automated way is an extra bonus.

As far as integrating efficient decision procedures with proof assistants is concerned, using reflection is both not new and not the only possibility. We have already mentioned the work of Samuel Boutin [Bou97], but the idea of reflection in theorem proving predates it, and was already used in Richard Weyhrauch's FOL system [Wey80]. The idea is the same: replace deductions by computations at the meta-level, and reflect this by allowing the system to accept meta-level computations as actual proofs, thereby taking shortcuts in proofs. This idea goes a bit further in Coq, in that there is no real separation in levels, as the proof language already includes a programming language.

Then, there are many ways to implement reflection. For example, S. Boutin [Bou97] distinguishes between total and partial reflection. *Partial reflection* consists in implementing and proving the intended algorithm inside the prover at hand. *Total reflection* additionally provides the reflection architecture a full description of the data structures used by the prover itself. With total reflection in Coq, it would be possible to extract a BDD algorithm to Caml, the implementation language of Coq, and have Coq actually run and still trust the extracted program. As of today, total reflection is not available in Coq, and our BDD algorithm will have to run in the interpreted lambda-calculus that is the logical core of Coq, instead of in compiled Caml. This will make it slower than what it could be. Total reflection has been proposed for Nuprl as well [BC93], and has been implemented at least once[1].

[1] James Caldwell, private communication.

Reflection also comes under many flavours. While S. Boutin does all the computations in an external program written in Caml, and on success has Coq redo all the computations a second time, Benjamin Werner[2] keeps a trace of the computation path that the external program followed, and has Coq replay only the computation steps which actually succeeded in producing a proof; this is interesting as computation in Coq is much slower than computation done by an external program. The latter method applies mostly to non-deterministic decision algorithms, and it does not seem easy to use such a strategy for BDDs. Maartijn Oostdijk [Oos96] has a different approach, where a separate language for specifying a large class of decidable predicates is used, and M. Oostdijk's proof calculator program generates a decision algorithm, a proof of correctness and converts the predicate, the algorithm and the proof to Lego objects, where the algorithm decides the predicate by partial reflection.

Now, reflection is not the only way to integrate model-checking with theorem proving, and there have been several publications on the subject already. First, it is precisely one of the main aims of PVS [ORS92] to do so; but the internal model-checker of PVS is itself not checked formally, for now at least. One of the closest works to ours in the literature is Shen-Wei Yu and Zhaohui Luo's paper on LegoMC [YL97]. They build a model-checker, LegoMC, for checking μ-calculus formulae against CCS programs; on success, LegoMC returns a proof-term expressing why it believes the given formula to hold on the given program. Safety is achieved through Lego checking the latter proof-term. (Please consult this paper for further references.) Interesting as it may be for simple verification conditions, this technique does not scale up much to more complex hardware circuit problems. In fact, for such problems, BDDs are mostly the only choice, and we have already seen that, although Harrison had managed to interface BDDs with HOL in such a way [Har95], the results were not entirely satisfactory.

Finally, we should mention that modelling BDDs in Coq has already been done, by Emmanuel Ledinot [Led93]. For the sake of simplicity, BDDs were actually modelled as binary decision *trees*, and sharing was simply ignored. As the goal of this development was to prove formally that the BDDs of two equivalent formulae were identical, it was reasonable to do so. However, in a real implementation, sharing is crucial, both in the data structures and in the algorithms, which is why we had to reimplement our BDDs from the ground up.

2 Binary Decision Diagrams

Consider a language of propositional formulae F, G, ..., built on a set of *propositional variables* A, B, C, ...:

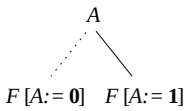
$$F ::= A \mid F \wedge F \mid F \vee F \mid F \Rightarrow F \mid \neg F$$

with its usual Tarskian semantics.

Let $\mathbf{1}$ (truth) and $\mathbf{0}$ (falsehood) be two new symbols, denoting true and false respectively. Shannon's decomposition principle states that for every propositional formula F, and every propositional variable A:

$$F \Leftrightarrow \left\{ \begin{array}{c} A \Rightarrow F[A := \mathbf{1}] \\ \neg A \Rightarrow F[A := \mathbf{0}] \end{array} \right. \tag{1}$$

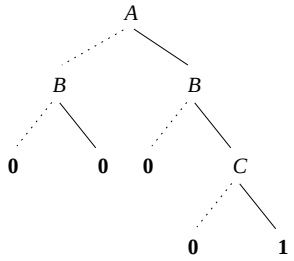
where the big curly brace denotes conjunction, and $F[A := F']$ is the result of replacing A by F' in F. We shall also use the convenient Prolog-like notation $F = A \longrightarrow F[A := \mathbf{1}]; F[A := \mathbf{0}]$, or the graphical notation



where the dashed line is the "A false" direction and the solid line is the "A true" direction.

Note that A does not appear any more in either $F[A := \mathbf{1}]$ or $F[A := \mathbf{0}]$. Choosing a propositional variable A occurring in a formula F, decomposing it with respect to A, and continuing the process on $F[A := \mathbf{1}]$ and $F[A := \mathbf{0}]$ recursively builds a binary tree (a *Shannon tree*) whose nodes are labeled by propositional variables and whose leaves are $\mathbf{1}$ or $\mathbf{0}$. For example, doing this on the formula $B \wedge (A \vee \neg B) \wedge (\neg A \vee C)$ yields the following Shannon tree:

[2] Of the Coq team; private communication.

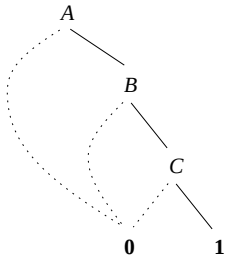


BDDs are *reduced*, *shared* and *ordered* versions of Shannon trees. We now explain what this all means.

Sharing is an implementation concern, and means that any two isomorphic subtrees of the Shannon tree are stored at the same address in memory, thus making a DAG (directed acyclic graph) from the tree, and saving some memory. Sharing can be accomplished by remembering all previously built BDDs in a global hash-table, so as to retrieve a previously constructed node instead of allocating memory for a new one with the same propositional variable and the same sons. Sharing is in fact the main reason why BDDs can be used at all, whereas unshared Shannon trees are in most cases much too large to fit in memory: the amount of space saved, though possibly close to nil, is usually quite large in practical applications of propositional logic. Modeling and implementing sharing will be one of our main concerns.

On the other hand, *reducing* is a logical operation. Reducing means converting every sub-DAG of the form $A \longrightarrow F; G$, where $F = G$, into F. This is possible because we are working in classical logic, so that the law of the excluded middle holds. Indeed, by the latter, $(A \Rightarrow F) \wedge (\neg A \Rightarrow F) \Leftrightarrow F$, that is, $A \longrightarrow F; F \Leftrightarrow F$.

The BDD corresponding to the example Shannon graph above is then:



This prompts us to the following definitions:

Definition 1 (Shannon Graph) *The set of* Shannon graphs *is the smallest containing* $\mathbf{1}$, $\mathbf{0}$, *and such that for every propositional variable* A, *for all Shannon graphs* F *and* G, *the triple* (A, F, G), *written* $A \longrightarrow F; G$, *is a Shannon graph.*

In the definition, Shannon graphs can be seen as DAGs, whose leaves are $\mathbf{1}$ or $\mathbf{0}$, and whose non-leaf nodes have one label A and two successors F and G.

Definition 2 (Reduction) *A Shannon graph is said to be* reduced *if it does not contain any subgraph of the form* $A \longrightarrow F; F$.

It remains to explain what it means for a BDD to be an *ordered* reduced Shannon graph. Observe that if we decompose formulas by selecting propositional variables in a given order, then reduced Shannon graphs are canonical forms for classical propositional formulas, called binary decision diagrams or BDDs. (This is not entirely obvious, and will have to be proved formally as well.) That is, if we fix a total strict ordering $<$ on propositional variables, and insist that we decompose formulas in increasing order of variables, we get BDDs where A is always less than any other kernel in $A \longrightarrow F; G$, and this ensures uniqueness of representation.

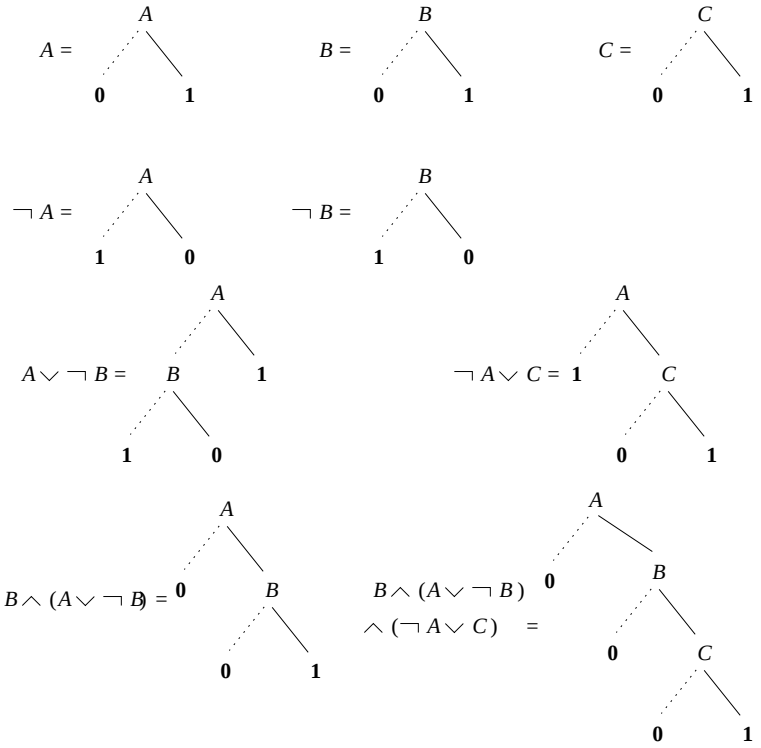
Such BDDs are sometimes called reduced ordered BDDs (see [Bry86]), but we shall just call them BDDs.

One of the reasons why BDDs are interesting is that they are usually compact. But BDDs are also interesting because they are easy to work with, at least in principle: all the usual logical operations are easily definable by recursive descent on BDDs. Negation, for example, is defined by $\neg\mathbf{1} =_{\mathrm{df}} \mathbf{0}$, $\neg\mathbf{0} =_{\mathrm{df}} \mathbf{1}$, $\neg(A \longrightarrow F; G) =_{\mathrm{df}} A \longrightarrow \neg F; \neg G$, and all other Boolean operations $\oplus$ ($\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, etc.) are defined by

$(A \longrightarrow F_1; G_1) \oplus (A \longrightarrow F_2; G_2) =_{\mathrm{df}} A \longrightarrow (F_1 \oplus F_2); (G_1 \oplus G_2)$, with the usual definitions on **1** and **0**. This is called *orthogonality* of $\oplus$.

By remembering in a cache the results of previous computations, negation can be computed in linear time, and binary operations in quadratic time w.r.t. the sizes of the input BDDs. (This is sometimes called *memoizing* or *tabulating*.) These times are given assuming a functional style of programming. Using side-effects provides for constant-time negation [Pos93], but well-known refinements of BDDs also provide for constant-time negation in functional style [Bil87]. Be aware, however, that although conjunction, for example, takes quadratic worst-case time in the size n of the two conjuncts, the result of the conjunction may be of size up to n^2. Then, a sequence of $f(n)$ binary operations may produce a BDD of size $n^{f(n)}$: when we say that BDDs have proved to be compact and efficient in some domains, we cannot claim they are a panacea and solve all propositional problems in polynomial time.

To build a BDD for a propositional formula, the naïve way would be to build a Shannon tree, to share it, to reduce it, and to permute kernels so as to eventually produce the desired BDD. This is grossly inefficient, and the preferred way of building BDDs from formulas is bottom-up. For example, to build the BDD for $A \wedge \neg B$, we first build the BDD of A and the BDD of B; from the latter, we compute that of $\neg B$; together with the former, we can then compute that of $A \wedge \neg B$. We invite the reader to redo the computations leading to the BDD of $B \wedge (A \vee \neg B) \wedge (\neg A \vee C)$ that we have already shown above. As a help, here are the intermediate BDDs (for legibility, we won't actually show the leaves **0** and **1** as shared):



3 A Short Tour of Coq

Coq is a proof assistant based on the Calculus of Inductive Constructions (CIC), a type theory that is powerful enough to formalize most of mathematics. It properly includes higher-order intuitionistic logic, augmented with definitional mechanisms for inductively defined types, sets, propositions and relations. CIC is also a typed λ-calculus, and can therefore also be used as a programming language. For a gentle introduction, see [HKPM98]. We shall describe here the main features of Coq that we shall need later.

The *sorts* of CIC are Prop, the sort of all propositions; Set, the sort of all specifications, programs and data types; and Type_i, $i \in \mathbb{N}$, which we won't need. The typing rules for sorts include $\mathrm{Prop} : \mathrm{Type}_0$, $\mathrm{Set} : \mathrm{Type}_0$, $\mathrm{Type}_i : \mathrm{Type}_{i+1}$.

What it means for Prop to be the sort of propositions is that any object F : Prop (read: F of type Prop) denotes a proposition, i.e., a formula. If F and G are propositions, F `->` G is a proposition (read: "F implies G"), and (`x` :T)G is a proposition, for any type T (read: "for every `x` of type T, G"). In turn, any object $\pi : F$ is a *proof* π of F. A formula is considered proved in Coq whenever we have succeeded to find a proof of it. Proofs are written, at least internally, as λ-terms, but we shall be content to know that we can produce them with the help of *tactics*, allowing one to write proofs by reducing the goal to subgoals, and eventually to immediate, basic inferences.

Similarly, any object t : Set denotes a data type, like the data type `nat` of natural numbers, or the type `nat -> nat` of all functions from naturals to naturals. In general, data types can be even more precise, like (`n` : `nat`){`m` : `nat`|`n` = (2) * `m`\/`n` = (2) * `m` + (1)}: this is the data type of functions taking a natural number `n` and returning some natural number `m` such that `n` is either twice `m` or twice `m` plus one, plus a proof that it is actually so. We will never use any data type as complex as this. Data types can, and will usually be, defined inductively. For instance, the standard definition of the type `nat` of natural numbers in Coq reads:

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

which means that `nat` is of type Set, hence is a data type, that `0` (zero) is a natural number, that `S` is a function from naturals to naturals (successor), and that every natural number must be of the form (`S` (`S` ...(`S` `0`)...)) (induction).

Then, if t : Set, and p : t, we say that p is a *program* of type t. Programs in Coq are pure functional programs: the language of programs is based on a λ-calculus with variables, application (`p q`) of `p` to `q` (in general, (`p` q_1 ... q_n) will denote the same as (...(`p` q_1)...q_n)), abstraction [`x` : `t`]`p` (where `x` is a variable; this denotes the function mapping `x` of type `t` to `p`), case splits (e.g., `Cases n of 0 => v | (S m) => (f m) end` either returns `v` if the natural number `n` is zero (`0`) or (`f m`) if `n` is the successor of some integer `m`), and functions defined by structural recursion on their last argument. For the latter, consider the following definition in Coq's standard prelude:

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

This is the standard definition of addition of natural numbers, by primitive recursion over the first argument. Notice the recursion is forced to take place on the first argument, by having the `plus` function actually return a function mapping its second argument `m` to the desired sum. For soundness reasons, Coq refuses to accept any non-terminating function. In fact, Coq refuses to acknowledge any `Fixpoint` definition that is not primitive recursive, even though its termination might be obvious.

In general, propositions and programs can be mixed. We shall exploit this mixing of propositions and programs in a very limited form: when π and π' are programs (a.k.a., descriptions of data), then π=π' is a proposition, expressing that π and π' have the same value. Reflection can then be implemented as follows: given a property P : t `->` `Prop` (say, P takes a propositional formula and expresses whether it is valid, where t is some data type of propositional formulae), write a program π : t `->` `bool` (say, π will build a BDD for the input formula, and return `true` if the BDD is `1`, `false` otherwise; note that `bool` is defined by `Inductive bool : Set := true : bool | false : bool.` and should be distinguished from `Prop`). Then prove the correctness lemma:

$$(x : t)\ (\pi\ x) = \texttt{true} \texttt{ -> } (P\ x)$$

To show that P holds of some concrete data x_0 (without any free Coq variable), you can then compute (π x_0) in Coq's *programming language*. If the result is `true`, then (π x_0)=`true` holds, and the correctness lemma then allows you to conclude that (P x_0) holds in Coq's *logic*. Our use of reflection will follow these lines, while not being exactly identical.

We have only presented a small part of Coq. The actual language of Coq is richer, as it allows more forms of inductive definitions, in particular. The interested reader may find a more formal and more complete description of Coq in [BBC+99].

The development of BDDs in Coq will also rest on several predefined libraries of proofs and code. This includes the theories of Peano arithmetic and Booleans, but also the second author's map library [GL98]. This is a theory of finite sets and maps, and their implementation in Coq as radix-exchange tries [Knu73], following closely the implementation in [Gou94]. The fundamental datatypes are:

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

which is intended to represent the presence of some data a : `A` (this is `(SOME A a)`, of type `(option A)`), or the absence of any data of type `A` (this is `(NONE A)`, again of type `(option A)`); and:

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

which is a type of *addresses*. The `ad_z` constant denotes 0, while `(ad_x p)` denotes the strictly positive number p. The `positive` data type is defined as part of the `ZARITH` standard Coq theory of arithmetic in $\mathbb{Z}$, and is defined as:

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

The intended semantics is that `xH` denotes 1, `xO` is multiplication by 2 (appending a 0 bit at the end of a number, in binary), and `xI` is $x \mapsto 2x + 1$ (appending a 1 bit at the end of x in binary). Elements of `ad` are viewed either as finite lists of bits (in a contorted way) or as natural numbers (in an unusual format).

Equality on addresses is decidable, and there is a function `ad_eq` of type `ad -> ad -> bool` such that `(ad_eq a a')` equals `true` if `a`=`a'`, and `false` otherwise. Similarly, `ad_le : ad -> ad -> bool` is such that `(ad_le a a')` returns `true` whenever `a`$\leq$`a'`, and `false` otherwise.

The maps of [GL98] always map addresses (of type `ad`) to elements of some type `A`. Given `A : Set`, the type of maps from `ad` to `A` is `(Map A)`. The following functions are provided, which we shall be interested in:

- `(M0 A)` or `(newMap A)` is the empty map from `ad` to `A`.
- `MapGet : (A:Set)(Map A)->ad->(option A)`; `(MapGet A` m x`)` reads the image y of x under the map m, and either returns `(SOME A` y`)` if it exists, or else `(NONE A)`; we also use this function to define the *semantics* of a map m as the function `(MapGet A` m`)`. The latter is then an encoding as a total function f of type `ad -> (option A)` of the partial function from `ad` to `A` mapping every x such that $f(x)$ is of the form `(SOME A` y`)` to y.
- `MapPut : (A:Set)(Map A)->ad->A->(Map A)`; `(MapPut A` m x y`)` adds the binding from x to y to the map m, and returns the new map. The newly added binding hides any previous binding of x in m. Semantically, it returns the partial function mapping x to y, and every $z \neq x$ to $m(z)$.
- `in_dom : (A:Set)ad->(Map A)->bool`; `(in_dom A` x m`)` checks whether $m(x)$ is defined, i.e., it returns `true` if and only if `(MapGet A` m x`)` does not return `(NONE A)`.

Note that, although we have to provide the type `A` explicitly in `(MapPut A m x y)`, it is also possible to let Coq infer it, by replacing the required type by a question mark, as in `(MapPut ? m x y)`.

Finite maps can be built element by element using `MapPut`. The map library contains also a few other useful functions on maps and generic iterators over finite maps, which we won't use, and a collection of theorems on maps, which we shall mention when we need them. Finite sets of addresses are considered to

be just maps from `ad` to the one-element type `unit`, that is, sets are represented as the domains of maps with trivial codomain. Finally, all these functions are defined as programs, they are not just axiomatized; therefore they provide a running implementation of finite maps and sets in Coq.

4 Implementing BDDs in Coq

4.1 Representation of BDDs

We use maps to model a state of the memory in which all the BDDs are stored.

Definition 3

```
Definition BDDvar := ad.
```

Definition 4

```
Definition BDDstate := (Map BDDvar * ad * ad).
```

BDD nodes are represented by addresses. At each address representing an internal node (i.e. different from the leaf nodes) we store the variable and the addresses of the left and right nodes. To implement sharing of isomorphic BDDs we have a sharing map which stores the address in the memory (represented by `BDDstate`) corresponding to a tuple of variable and left and right addresses. Since we have a single `BDDstate` in which all the BDDs are stored, it allows us not just to share the common subgraphs of one BDD, we also share the common sub-BDDs of two or more different BDDs. Our configuration consists of a `BDDstate`, a `BDDsharing_map`, and a `counter`, which is again of type `ad`. The latter counter is used to represent the highest unallocated address in memory; we indeed use a very simple stack-like allocation scheme, as the emphasis of this work is not on memory management.

Definition 5

```
Definition BDDsharing_map := (Map (Map (Map ad))).
```

Definition 6

```
Definition BDDconfig := BDDstate * BDDsharing_map * ad.
```

We have used the same type `ad` to represent variables. This makes it easier to define sharing maps using the map library. Sharing maps need to map a tuple $(x, (l, r))$ of variable and left and right nodes to an address. However the map library only allows maps from `ad` to some other type. This problem is solved by defining sharing maps as maps from left node l to maps from right node r to maps from variables x to addresses. This requires the variable x to be of type `ad` as well, whence Definition 3.

Also since `ad` is just a representation of natural numbers, it allows us to number the variables. It also gives a simple ordering on the variables as the standard ordering on natural numbers. The leaves **0** and **1** are always in the first two locations in the memory. We do not store anything in these two locations.

Definition 7

```
Definition BDDzero := ad_z.
```

Definition 8

```
Definition BDDone := (ad_x xH).
```

It is convenient to have the same type `ad` for the leaf nodes. Having special representation for leaf nodes would make it difficult to define the sharing maps and would complicate the rest of the functions. It is then necessary to reserve two locations for the leaf nodes so that nothing else is stored there. The first two locations are then the natural choice. It also makes allocation of nodes convenient. It works just by storing the new node at the address pointed to by the counter and then incrementing the counter.

To ensure that a configuration indeed stores BDDs and that they are reduced, shared and ordered we define some conditions when a configuration is OK. The conditions are as follows :

1. There is no tuple stored in the first two locations of the `BDDstate` as they represent leaf nodes.
2. If there is any tuple $(x,(l,r))$ stored at any address, then l and r are distinct (reduced-ness condition), and the variables stored at the addresses l and r are strictly less than x (ordering condition.) These are ensured by the `BDDbounded` predicate defined below on each node, and by `BDDstate_OK` on all nodes, together with the first condition.
3. To ensure shared BDDs we have the predicate `BDDsharing_OK` which ensures that the sharing map is compatible with BDD state.
4. Finally we put some condition on the BDDs to ensure that allocation of new nodes in the memory works properly. The conditions are that the counter is greater than 1 and that there is no node stored after the counter.

Definition 9

```
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).
```

`BDDbounded` is defined as an inductive predicate, that is, as clauses: `(BDDbounded bs node n)` ("`node` is bounded by `n` in state `bs`") is intended to hold when node `node` is the root of a reduced and ordered BDD in state `bs`, whose variables are all strictly less than `n`. This is always the case when `node` is **0** (clause `BDDbounded_0`) or **1** (clause `BDDbounded_1`). Otherwise, this is the case when `node` is mapped by the state `bs` to some triple (x,l,r) with $x < n$, $l \neq r$, and l and r bounded by x (clauses `BDDbounded_2`). `BDDcompare` is the function which compares two variables and returns an element of type `relation` which consists of the elements `EGAL` (equal), `INFERIEUR` (less than) and `SUPERIEUR` (greater than); it is build from the auxiliary function `compare` in `ZARITH`:

Definition 10

```
Definition BDDcompare := [x,y:BDDvar]
  Cases x y of
      ad_z ad_z => EGAL
    | ad_z (ad_x _) => INFERIEUR
    | (ad_x _) ad_z => SUPERIEUR
    | (ad_x p1) (ad_x p2) => (compare p1 p2 EGAL)
  end.
```

The variables on any path in the graph are assumed to be in decreasing order. This is contrary to the usual convention. However it allows us to use the fact that $<$ is well-founded on `ad`, so the conditions defined

in BDDbounded additionally imply that any bounded BDD node is the root of a *finite acyclic* graph. This would have been a tricky condition to enforce without the convention that every BDD node is bounded.

This choice of ordering also gives a natural induction principle for doing all the proofs related to BDDs. Most of the proofs use well founded induction on the given bound of the BDD node.

Definition 11

```
Definition BDDordered := [bs:BDDstate; node:ad]
  Cases (MapGet ? bs node) of
      NONE => True
    | (SOME (n, _)) => (BDDbounded bs node (ad_S n))
  end.
```

Definition 12

```
Definition BDD_OK := [bs:BDDstate; node:ad]
  (BDDordered bs node).
```

Conditions 1 and 2 are enforced by the following:

Definition 13

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

Condition 3 on sharing is expressed as follows, where BDDshare_lookup is defined below. It just says that the sharing map and the state are, up to isomorphism, inverse maps.

Definition 14

```
Definition BDDsharing_OK := [bs:BDDstate; share:BDDsharing_map]
      (x:BDDvar) (l,r:ad) (a:ad)
            (BDDshare_lookup share x l r)=(SOME ? a) <->
            (MapGet ? bs a)=(SOME ? (x, (l, r))).
```

Now a configuration consisting of a state bs, a sharing map share and an allocation counter counter is OK if and only if the state is OK, the sharing map is OK with respect to the state, no address is allocated at counter or above, and counter is at least 2:

Definition 15

```
Definition BDDconfig_OK := [cfg : BDDconfig]
    <Prop>Cases cfg of
        (bs, (share, counter)) =>
          (BDDstate_OK bs) /\ (BDDsharing_OK bs share) /\
          ((a:ad) (ad_le counter a)=true -> (MapGet ? bs a)=(NONE ?))
          /\ (ad_le (ad_S (ad_S ad_z)) counter)=true
    end.
```

ad_S is the function which takes an address and returns the next address. The semantics of ad_S in terms of natural numbers is stated in the following lemma.

Lemma 1

```
Lemma ad_S_is_S : (a:ad)(nat_of_ad (ad_S a))=(S (nat_of_ad a)).
```

`BDDshare_lookup` looks up a sharing map for a tuple of variable and left and right nodes and returns the node containing it, if found.

Definition 16

```
Definition BDDshare_lookup : BDDsharing_map->BDDvar->ad->ad->(option ad) :=
   [share: BDDsharing_map;x:BDDvar;l,r:ad]
     Cases (MapGet ? share l) of
         NONE => (NONE ad)
       | (SOME m1) =>
           Cases (MapGet ? m1 r) of
               NONE => (NONE ad)
             | (SOME m2) =>
                 Cases (MapGet ? m2 x) of
                     NONE => (NONE ad)
                   | (SOME y) => (SOME ad y)
                 end
           end
     end.
```

Insertions into the sharing map are done using the function `BDDshare_put` defined below. The difficulty is that sharing maps are curried: to find the entry corresponding to (x, l, r) in the sharing map `share`, we must first look l up in `share`; this returns a map `m1`, where we must look up r; this returns a map `m2`, to which we add the binding from x to `counter` (the address of the node we shall allocate next); this gives a new map `m'2`, then we must add to `m1` the binding from r to `m'2`; this gives a new map `m'1`, and the result is `share` with the binding from l to `m'1` added:

Definition 17

```
Definition BDDshare_put : BDDsharing_map->BDDvar->ad->ad->ad->BDDsharing_map
       := [share:BDDsharing_map;x:BDDvar;l,r,counter:ad]
    let m1 = Cases (MapGet ? share l) of
                 (SOME y) => y
               | NONE => (newMap (Map ad))
             end in
    let m2 = Cases (MapGet ? m1 r) of
                 (SOME y) => y
               | NONE => (newMap ad)
             end in
    let m2' = (MapPut ? m2 x counter) in
    let m1' = (MapPut ? m1 r m2') in
        (MapPut ? share l m1').
```

We also have the functions `var`, `high` and `low` which respectively return the variable, the right node and the left node stored at a particular address.

Definition 18

```
Definition var := [cfg:BDDconfig; node:ad]
  Cases (MapGet ? (Fst cfg) node) of
      NONE => (* Error *) ad_z
    | (SOME (x,(l,r))) => x
  end.
```

Definition 19

```
Definition low := [cfg:BDDconfig; node:ad]
  Cases (MapGet ? (Fst cfg) node) of
      NONE => (* Error *) BDDzero
    | (SOME (x,(l,r))) => l
  end.
```

Definition 20

```
Definition high := [cfg:BDDconfig; node:ad]
  Cases (MapGet ? (Fst cfg) node) of
      NONE => (* Error *) BDDzero
    | (SOME (x,(l,r))) => r
  end.
```

Using the above definitions, we can trivially prove the ordering property of BDDs.

Lemma 2 *If the left node l of an internal node n is again an internal node, then the variable stored at l is less than the variable stored at n. Similarly, if the right node r of an internal node n is again an internal node, then the variable stored at r is less than the variable stored at n. In Coq notation:*

```
Lemma BDDvar_ordered_low : (cfg:BDDconfig)(node:ad)
   (BDDconfig_OK cfg) -> (is_internal_node cfg node) ->
   (is_internal_node cfg (low cfg node)) ->
   (BDDcompare (var cfg (low cfg node)) (var cfg node))=INFERIEUR.

Lemma BDDvar_ordered_high : (cfg:BDDconfig)(node:ad)
   (BDDconfig_OK cfg) -> (is_internal_node cfg node) ->
   (is_internal_node cfg (high cfg node)) ->
   (BDDcompare (var cfg (high cfg node)) (var cfg node))=INFERIEUR.
```

where `is_internal_node` is the condition that there is some tuple stored at the node. The lemma follows immediately from the conditions in `BDDbounded`, which say that l and r are bounded by the variable stored at *node*. This in turn implies that the variables stored at l and r are less than the variable at *node*.

We define the condition when is a node said to be OK in a configuration. Note that a node could be an arbitrary address in the memory, but all those addresses do not necessarily represent BDDs. For example, we do not store any node at an address greater that the counter. A node is OK in a configuration if it is a leaf node or there is a tuple $(x,(l,r))$ stored at that address.

Definition 21

```
Definition node_OK := [bs:BDDstate] [node:ad]
   node=BDDzero \/ node=BDDone \/ (in_dom ? node bs)=true.

Definition config_node_OK := [cfg:BDDconfig](node_OK (Fst cfg)).
```

Lemma 3 *The left and right nodes of an internal node are OK nodes in a configuration, provided that the configuration is OK.*

```
Lemma high_OK : (cfg:BDDconfig)(node:ad)
  (BDDconfig_OK  cfg) ->
  (is_internal_node cfg node) ->
  (config_node_OK cfg (high cfg node)).

Lemma low_OK : (cfg:BDDconfig)(node:ad)
```

```
  (BDDconfig_OK  cfg) ->
  (is_internal_node cfg node) ->
  (config_node_OK cfg (low cfg node)).
```

4.2 Interpretation of BDDs

We define an interpretation of BDD nodes in a configuration as boolean functions.

Definition 22 (Boolean function) *Let an* environment *be a map from variables to booleans, and a* boolean function *be a function mapping environments to booleans. In Coq notation:*

```
Definition var_binding := BDDvar->bool.
Definition bool_fun := var_binding->bool.

Definition bool_fun_zero := [vb:var_binding]false.
Definition bool_fun_one := [vb:var_binding]true.
```

`bool_fun_zero` and `bool_fun_one` respectively represent the boolean functions that are always false and always true.

Equality on boolean functions is defined as follows :

```
Definition bool_fun_eq := [bf1,bf2:bool_fun]((vb:var_binding) (bf1 vb)=(bf2 vb)).
```

The main logical operations on boolean functions, negation, disjunction, conjunction, implication, double implication, are defined as follows:

```
Definition bool_fun_neg : bool_fun -> bool_fun
      := [bf:bool_fun]([vb:var_binding](negb (bf vb))).
Definition bool_fun_or : bool_fun -> bool_fun -> bool_fun
     := [bf1,bf2:bool_fun]([vb:var_binding](orb (bf1 vb) (bf2 vb))).
Definition bool_fun_and : bool_fun -> bool_fun -> bool_fun
     := [bf1,bf2:bool_fun]([vb:var_binding](andb (bf1 vb) (bf2 vb))).
Definition bool_fun_impl : bool_fun -> bool_fun -> bool_fun
     := [bf1,bf2:bool_fun]([vb:var_binding](implb (bf1 vb) (bf2 vb))).
Definition bool_fun_iff : bool_fun -> bool_fun -> bool_fun
     := [bf1,bf2:bool_fun]([vb:var_binding](eqb (bf1 vb) (bf2 vb))).
```

where `negb`, `orb`, `andb` and `eqb` are the corresponding logical operations on booleans.

We now wish to define the function `bool_fun_of_BDD` which returns the boolean function represented by a given node in a given configuration.

The function is usually defined by structural recursion on the graph of the BDD: calling it f temporarily for short, $f(\mathbf{0})$ is false, $f(\mathbf{1})$ is true, and $f(A \longrightarrow F; G)$ is "if A then $f(F)$ else $f(G)$", up to some abuse of notation. However, since BDDs are represented as maps, there is no direct way of applying recursion. Note that in Coq, only recursive functions with structural recursion on the last argument are accepted. So we supply an extra argument of type `nat` and define the function by recursing on this argument. We need to ensure that this argument is greater than the maximum depth of recursion (in this case, the depth of the BDD).

Since the variables on all the paths are in decreasing order, the depth of recursion is at most the number of the variable stored at the root node. Thus the required bound is easily calculated.

Definition 23

```
Fixpoint bool_fun_of_BDD_1 [cfg:BDDconfig;node:ad;bound:nat] : bool_fun :=
  Cases (MapGet ? (Fst cfg) node) of
      NONE => if (ad_eq node BDDzero) then bool_fun_zero else bool_fun_one
    | (SOME (x,(l,r))) =>
```

```
      Cases bound of
          O => (* Error *) bool_fun_zero
        | (S bound') =>
            let bfl = (bool_fun_of_BDD_1 cfg l bound') in
            let bfr = (bool_fun_of_BDD_1 cfg r bound') in
                [vb:var_binding](if (vb x) then (bfr vb) else (bfl vb))
      end
  end.

Definition bool_fun_of_BDD := [cfg:BDDconfig;node:ad]
     (bool_fun_of_BDD_1 cfg node (S (nat_of_ad (var cfg node)))).
```

We require to prove that the function `bool_fun_of_BDD_1` does not depend on the last argument (bound). This is stated in the following lemma :

Lemma 4 *The semantics* `(bool_fun_of_BDD_1 cfg node bound)` *of the BDD node* `node` *in configuration* `cfg` *is independent of the bound* `bound` *provided it is high enough. In Coq:*

```
Lemma bool_fun_of_BDD_1_change_bound : (cfg:BDDconfig) (BDDconfig_OK cfg) ->
  (bound:nat)(node:ad)
  (lt (nat_of_ad (var cfg node)) bound) ->
  (bool_fun_of_BDD_1 cfg node bound)
   =(bool_fun_of_BDD_1 cfg node (S (nat_of_ad (var cfg node)))).
```

Proof: We prove the lemma by well founded induction on `bound`. If `node` is a leaf node then the corresponding boolean function is either `bool_fun_zero` or `bool_fun_one` which is clearly independent of bound. If `node` is an internal node containing the tuple `(x,(l,r))` then by induction hypothesis, we prove that the boolean function corresponding to `l` and `r` are independent of bound. Here we require lemmas `BDDvar_ordered_high` and `BDDvar_ordered_low` from which we get that the variables stored at `l` and `r` are less that `x`. We need to separately consider the cases where `l` and `r` are leaf nodes or internal nodes, since in the lemmas `BDDvar_ordered_high` and `BDDvar_ordered_low` we require `r` and `l` to be internal nodes. □

Using the above lemma, we get the following required property of `bool_fun_of_BDD`.

Lemma 5 *The function* `bool_fun_of_BDD` *interprets the leaf node* `BDDzero` *as the boolean function* `bool_fun_zero` *and* `BDDone` *as* `bool_fun_one`. *An internal node containing the tuple* `(x,(l,r))` *is interpreted as the boolean function* `if x then bf1 else bf2` *where* `bf1` *and* `bf2` *are the interpretations of* `l` *and* `r`. *In Coq:*

```
Lemma bool_fun_of_BDD_semantics : (cfg:BDDconfig)
   (BDDconfig_OK cfg) ->
   (bool_fun_of_BDD cfg BDDzero) = bool_fun_zero
    /\ (bool_fun_of_BDD cfg BDDone) = bool_fun_one
    /\ ((node:ad)(is_internal_node cfg node) ->
        (bool_fun_of_BDD cfg node)
          =([vb:var_binding]
             (if (vb (var cfg node))
              then
                ((bool_fun_of_BDD cfg (high cfg node)) vb)
             else
                ((bool_fun_of_BDD cfg (low cfg node)) vb)))).
```

Since `bool_fun_of_BDD` is defined in terms of `bool_fun_of_BDD_1` the proof of the lemma makes use of the previous lemma that `bool_fun_of_BDD_1` is independent of the bound supplied as argument.

In the sequel, we shall need an additional property of `bool_fun_of_BDD`, *extensionality*: say that a boolean function is extensional if and only if it maps equal environments to equal boolean values. Two

environments are deemed equal whenever they map each variable to identical boolean values. This may seem trivial, but functions in Coq are in general *not* extensional. We define the extensionality predicate `bool_fun_ext` as follows:

Definition 24

```
Definition bool_fun_ext := [bf:bool_fun]
  ((vb,vb':var_binding)
       ((x:BDDvar)(vb x)=(vb' x)) ->
          (bf vb)=(bf vb')).
```

Although not all boolean functions are extensional, all the boolean functions that we will be dealing with will be. We prove that the boolean functions returned by `bool_fun_of_BDD_1` and consequently by `bool_fun_of_BDD` satisfy the `bool_fun_ext` predicate.

Lemma 6

```
Lemma bool_fun_of_BDD_1_ext : (bound:nat)(cfg:BDDconfig)
  (node:ad)(bool_fun_ext (bool_fun_of_BDD_1 cfg node bound)).

Lemma bool_fun_of_BDD_ext : (cfg:BDDconfig)(node:ad)
  (bool_fun_ext (bool_fun_of_BDD cfg node)).
```

Proof: The first lemma is proved by a simple induction on `bound` and using the definition of the function `bool_fun_of_BDD_1`. The required lemma `bool_fun_of_BDD_ext` follows immediately. □

We can easily prove that the property `bool_fun_ext` of boolean functions is preserved by the logical operations of negation, disjunction, conjunction, implication and double implication on them.

4.3 Uniqueness of BDDs

Here we outline how we proved the uniqueness of BDDs for our implementation using the conditions in `BDDconfig_OK`. We proved that in a configuration that is OK, if two OK nodes have the same interpretation as boolean functions then they are stored at the same address.

We require to prove a few lemmas before that. While they appear to be quite obvious they require somewhat tedious proofs in Coq.

Lemma 7 *The interpretation of an internal node as boolean function is independent of variables which are greater than that stored at the node.*

```
Lemma BDDvar_independent_1 :
  (cfg:BDDconfig)(n:nat)(node:ad)(x:BDDvar)
  (BDDconfig_OK cfg) ->
  (is_internal_node cfg node) ->
  n=(nat_of_ad (var cfg node)) ->
  (BDDcompare (var cfg node) x)=INFERIEUR ->
  (bool_fun_independent (bool_fun_of_BDD cfg node) x).
```

Independence is defined as follows :

Definition 25 *A boolean function* `f` *is said to be independent of a variable* `x` *if in any environment,* `f` *evaluates to the same boolean irrespective of the value of* `x` *in that enviroment.*

```
Definition augment := [vb:var_binding;x:BDDvar;b:bool]
  ([y:BDDvar](if (ad_eq x y) then b else (vb y))).

Definition bool_fun_independent := [bf:bool_fun][x:BDDvar]
```

```
  ( (vb:var_binding)
    (bf (augment vb x true))
     = (bf (augment vb x false))).
```

The function `augment` modifies an environment by changing the value of some variable in that enviroment.

Proof: (of Lemma 7) The proof of `BDDvar_independent_1` proceeds by well founded induction on `n`, the natural number represented by the variable stored at `node`. Since the variables stored at the left and right nodes of `node` are less than the variable stored at `node` (from the lemmas `BDDvar_ordered_high` and `BDDvar_ordered_low`), they are also less than `x` (transitivity of `BDDcompare`). Thus the interpretation of the two sub-BDDs are independent of `x`. Now we apply `bool_fun_of_BDD_semantics` to get the boolean function corresponding to `node` and prove that it is independent of `x`. □

Lemma 8 *The interpretation of internal node as boolean function is not a constant boolean function.*

```
Lemma internal_node_not_constant_1 : (cfg:BDDconfig)(BDDconfig_OK cfg) ->
  (n:nat)(node:ad)
  (is_internal_node cfg node) ->
  n=(nat_of_ad (var cfg node)) ->
  ~(bool_fun_eq (bool_fun_of_BDD cfg node) bool_fun_zero)
  /\ ~(bool_fun_eq (bool_fun_of_BDD cfg node) bool_fun_one).
```

Proof: The proof works by well founded induction on `n`. We need to consider several cases. Let `(x,(l,r))` be the tuple stored at `node`. If both `l` and `r` are leaf nodes (i.e. either `BDDzero` or `BDDone`), then since `l` and `r` are distinct (see the conditions in `BDDbounded`), one of them will be interpreted as `bool_fun_zero` and the other as `bool_fun_one`. In this case the interpretation of `node` will be that of `x` or of its negation, so it cannot be a constant function. In case either of `l` and `r` is a non-leaf node, then from induction hypothesis it will not be a constant function, and consequently the interpretation of `node` will not be a constant function. Note that `l` and `r` are distinct in this case also, but we cannot say that the corresponding boolean functions are distinct as we have not yet proved the uniqueness of BDDs. □

We can now prove the required uniqueness lemma.

Lemma 9 *BDDs representing equivalent boolean functions are stored at the same location in a configuration.*

```
Lemma BDDunique : (cfg:BDDconfig)
  (BDDconfig_OK cfg) ->
  (node1,node2:ad)
  (config_node_OK cfg node1) ->
  (config_node_OK cfg node2) ->
  (bool_fun_eq (bool_fun_of_BDD cfg node1) (bool_fun_of_BDD cfg node2)) ->
  (node1=node2).
```

Proof: The proof requires well founded induction on the maximum of the variables stored at `node1` and `node2`. Note that this is well defined, since we took these variables to be 0 in case of leaf nodes (see `var`, Definition 18). We first prove the following auxilliary lemma, from which the required lemma will immediately follow.

```
Lemma BDDunique_1 : (cfg:BDDconfig)
  (BDDconfig_OK cfg) ->
  (n:nat)(node1,node2:ad)
  n=(max (nat_of_ad (var cfg node1)) (nat_of_ad (var cfg node2))) ->
  (config_node_OK cfg node1) ->
  (config_node_OK cfg node2) ->
  (bool_fun_eq (bool_fun_of_BDD cfg node1) (bool_fun_of_BDD cfg node2)) ->
  (node1=node2).
```

Introducing the universal quantification on `n` serves only as a trick to apply well founded induction on `n`. We consider several cases. If both `node1` and `node2` are leaf nodes (`BDDzero` or `BDDone`) then they can't be distinct since in that case one of them would be interpreted as `bool_fun_zero` and the other as `bool_fun_one` violating assumptions. If one of them is a leaf node then the other cannot be an internal node because that will violate the fact the interpretation of an internal node is not a constant function (Lemma 8).

Now we consider the case where both the nodes are internal. Let `x1` and `x2` be the variables stored at nodes `node1` and `node2` respectively. We have three cases, depending on `x1` being less than, equal to, or greater than `x2`.

If `x1` equals `x2`, then since the boolean functions that `node1` and `node2` denote are equal, the boolean functions corresponding to the two left nodes are also equal. Thus the two left nodes must be equal from induction hypothesis; the latter indeed applies, since the maximum of the variables stored at the two left nodes is less than `n`, provided these left nodes are not internal. Similarly we prove that the two right nodes are equal.

In case `x1` is less than `x2`, then the boolean function corresponding to `node1` will be independent of `x2`, by `BDDvar_independent_1` (Lemma 7). However the boolean function corresponding to `node2` is not independent of `x2` because the boolean functions corresponding to its left and right sub-BDDs are distinct, from induction hypothesis and the condition in `BDDbounded` that the two sub-nodes need to be distinct. Thus we get a contradiction. Similarly, `x2` cannot be less than `x1`.

Thus we have proved that the tuple stored at `node1` is same as that stored at `node2`. To show that `node1` is equal to `node2`, we use the sharing property of BDDs given by the `BDDsharing_OK` predicate. □

Ledinot already proved a similar property [Led93]: that any two binary decision trees (no sharing) that denote the same boolean function are structurally equal. We have decided not to reuse this result for two reasons. First, we need more than structural equality of the two BDDs: we require the actual equality of addresses where they are stored. Second, reusing Ledinot's result would have required that we prove some form of one-to-one correspondence between binary decision trees (with structural equality) and binary decision diagrams (with pointer equality, but up to an arbitrary permutation of addresses in the store). This did not seem easy at all.

Next we discuss the various operations on BDDs that we implemented. Since all the operations must change the state of the memory (by adding or deleting nodes) all of them take a configuration as argument and return a new configuration, in addition to other arguments and results. To ensure that we always work with configurations and nodes which are OK, we will prove that all operations on BDDs return configurations and nodes which are OK, provided the configurations and nodes in the arguments are OK.

4.4 Memory Allocation

First we define a simple allocator which allocates new nodes in a configuration. This is done by simply adding the tuple $(x, (l, r))$, which we want to allocate, at the location pointed to by the counter, and incrementing the counter. We also make a corresponding entry in the sharing map:

Definition 26

```
Definition BDDalloc := [cfg : BDDconfig; x : BDDvar; l,r : ad]
  Cases cfg of (bs, (share, counter)) =>
    let share' = (BDDshare_put share x l r counter) in
    let bs' = (MapPut ? bs counter (x,(l,r))) in
    let counter' = (ad_S counter) in
     ((bs',(share',counter')),counter)
  end.
```

To ensure that allocating nodes does not destroy the reducedness and sharing of BDDs, we define another function `BDDmake` which first looks up the sharing map for the tuple and allocates the node only if the tuple is not found there. Otherwise it returns the address where it is already stored. In this way sharing of BDDs is ensured. To ensure reducedness, it checks whether l is equal to r. If they are equal, then instead of allocating the node, it simply returns the address l.

Definition 27

```
Definition BDDmake := [cfg:BDDconfig; x:BDDvar; l,r:ad]
  if (ad_eq l r)
    then (cfg,l)
    else Cases cfg of
              (bs,(share,counter)) =>
                Cases (BDDshare_lookup share x l r) of
                    (SOME y) => (cfg,y)
                  | NONE => (BDDalloc cfg x l r)
                end
         end.
```

To prove the correctness of BDDmake we made the following assumptions on the arguments:

- The configuration `cfg` is OK.
- The nodes `l` and `r` are OK in cfg.
- The variable `x` is greater that the variables stored at l and r, provided they are internal nodes.

and proved the following results

- The new configuration returned is OK.
- The node returned is OK in the new configuration.
- All nodes of the old configuration are preserved in the new configuration, i.e. if there is a tuple stored at a node `node` in the old configuration, then the same tuple is also stored in the new configuration at `node`.
- The node returned by `BDDmake` contains the tuple `(x,(l,r))` in case `l` and `r` are distinct, otherwise `node` is the same as `l`.
- We proved an additional property that no other nodes are added in the configuration, which was however not required later on.

Lemma 10

```
Lemma BDDmake_semantics : (cfg:BDDconfig; l,r:ad; x:BDDvar)
  (BDDconfig_OK cfg)
  ->(node_OK (Fst cfg) l)
  ->(node_OK (Fst cfg) r)
  ->((xl:BDDvar; ll,rl:ad)
        (MapGet BDDvar*ad*ad (Fst cfg) l)
        =(SOME BDDvar*ad*ad (xl,(ll,rl)))
        ->(BDDcompare xl x)=INFERIEUR)
  ->((xr:BDDvar; lr,rr:ad)
        (MapGet BDDvar*ad*ad (Fst cfg) r)
         =(SOME BDDvar*ad*ad (xr,(lr,rr)))
         ->(BDDcompare xr x)=INFERIEUR)
  ->(BDDconfig_OK (Fst (BDDmake cfg x l r)))
     /\((ad_eq l r)=false
          ->(MapGet BDDvar*ad*ad (Fst (Fst (BDDmake cfg x l r)))
                                 (Snd (BDDmake cfg x l r)))
             =(SOME BDDvar*ad*ad (x,(l,r))))
     /\((ad_eq l r)=true->(Snd (BDDmake cfg x l r))=l)
```

```
    /\((a,l',r':ad; x':BDDvar)
       ((MapGet BDDvar*ad*ad (Fst (Fst (BDDmake cfg x l r))) a)
         =(SOME BDDvar*ad*ad (x',(l',r')))
         ->(MapGet BDDvar*ad*ad (Fst cfg) a)
            =(SOME BDDvar*ad*ad (x',(l',r')))
            \/(Snd (BDDmake cfg x l r))=a)
        /\((MapGet BDDvar*ad*ad (Fst cfg) a) =(SOME BDDvar*ad*ad (x',(l',r')))
            ->(MapGet BDDvar*ad*ad (Fst (Fst (BDDmake cfg x l r))) a)
               =(SOME BDDvar*ad*ad (x',(l',r')))))
    /\(node_OK (Fst (Fst (BDDmake cfg x l r))) (Snd (BDDmake cfg x l r))).
```

The main theorem required for the proof is `MapPut_semantics` given in the maps library.

```
Lemma 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')).
```

where `eqm` is the equality of two functions from addresses to some set.

```
Definition eqm = [A:Set; g,g':(ad->(option A))](a:ad)(g a)=(g' a).
```

Besides this, we required a similar property for `BDDshare_lookup` and `BDDshare_put` defined earlier.

Lemma 11

```
Lemma BDDshare_put_puts : (share:BDDsharing_map)(x:BDDvar)(l,r,a:ad)
  (BDDshare_lookup (BDDshare_put share x l r a) x l r)=(SOME ? a).
```

Lemma 12

```
Lemma BDDshare_put_preserves_nodes : (share:BDDsharing_map)(x,x':BDDvar)
  (l,l',r,r',a,a':ad)
  (BDDshare_lookup share x' l' r')=(SOME ? a') ->
  ~(x,(l,r))=(x',(l',r')) ->
  (BDDshare_lookup (BDDshare_put share x l r a) x' l' r')
  =(SOME ? a').
```

The previous two lemmas state the expected property of `BDDshare_lookup` and `BDDshare_put`, that `BDDshare_put` actually inserts the node into the sharing map and original entries in the sharing map are preserved, under suitable conditions (which are satisfied by our representation of BDDs). We then proved the semantics of `BDDmake` in terms of boolean functions.

Lemma 13 *The node returned by* `BDDmake` *is interpreted in the new configuration by the boolean function* `(bool_fun_if x bfr bfl)` *where* `bfr` *and* `bfl` *are respectively the interpretations of* `r` *and* `l` *in* `cfg`.

```
Lemma BDDmake_bool_fun : (cfg:BDDconfig; l,r:ad; x:BDDvar)
  (BDDconfig_OK cfg) ->
  (config_node_OK cfg l) ->
  (config_node_OK cfg r) ->
  ((is_internal_node cfg l) -> (BDDcompare (var cfg l) x)=INFERIEUR) ->
  ((is_internal_node cfg r) -> (BDDcompare (var cfg r) x)=INFERIEUR) ->
  (bool_fun_eq
    (bool_fun_of_BDD (Fst (BDDmake cfg x l r)) (Snd (BDDmake cfg x l r)))
    (bool_fun_if x (bool_fun_of_BDD cfg r) (bool_fun_of_BDD cfg l))).
```

Additionally, we proved that `BDDmake` preserves the interpretations of the nodes in the old configuration. This follows from the following lemma and from the fact that `BDDmake` preserves the nodes in the old configuration.

Definition 28

```
Definition nodes_preserved = [cfg,cfg':BDDconfig]
 (x:BDDvar; l,r,node:ad)
 (MapGet BDDvar*ad*ad (Fst cfg) node)=(SOME BDDvar*ad*ad (x,(l,r)))
 ->(MapGet BDDvar*ad*ad (Fst cfg') node)=(SOME BDDvar*ad*ad (x,(l,r))).
```

Lemma 14

```
Lemma nodes_preserved_3 : (cfg,cfg':BDDconfig; node:ad)
  (BDDconfig_OK cfg)
  ->(BDDconfig_OK cfg')
  ->(nodes_preserved cfg cfg')
  ->(config_node_OK cfg node)
  ->(bool_fun_eq (bool_fun_of_BDD cfg' node)
                 (bool_fun_of_BDD cfg node)).
```

`(nodes_preserved cfg cfg')` is proposition that all tuples stored in `cfg` are also stored in `cfg'` (at the same addresses). We have already proved that this is true when `cfg'` is the configuration that we get by applying `BDDmake` on `cfg`, with some conditions on the arguments supplied to `BDDmake`. `nodes_preserved_3` can be proved directly from `bool_fun_of_BDD_semantics` using well founded induction.

`BDDmake` is useful because of the fact that for applying it, we do not have to worry about whether the node already exists in the configuration or whether the nodes `l` and `r` are equal. However we still have to make sure that the variable `x` is greater than the variables stored at `l` and `r`. All the operations on BDDs, like disjunction, negation, etc. add new nodes in the configuration using `BDDmake`, which ensures that we always work with configurations and nodes that are OK.

4.5 Negation

Negation works by structural recursion on the graph of the BDD. To compute the negation of an internal node `node` containing the tuple `(x,(l,r))` in a configuration `cfg`, we first recursively apply negation on node `l` in the configuration `cfg` to get a new configuration `cfgl` and node `nodel`. Then we apply negation on the node `r` in the new configuration `cfgl` to get a configuration `cfgr` and node `noder`. Finally we apply `BDDmake` on the configuration `cfgr`, variable `x` and nodes `nodel` and `noder`. This is based on the orthogonality of negation as discussed in Section 2. The difference here is that we are working with configurations and we need to take into account the fact that applying any operation changes the configuration.

Also note that a naive implementation of negation using recursion can be expensive because of the fact negation of the same node may need to be computed several times. This can be prevented by caching, or *memoizing*, all the previously computed values of the function for various arguments. We first implemented and proved negation without caching, and then implemented negation with memoization and proved that it returns the same value as the original function.

4.5.1 Negation without Memoization

The following is the function to compute negation without memoization.

Definition 29

```
Fixpoint BDDneg_2 [cfg:BDDconfig; node:ad; bound:nat]
       : BDDconfig*ad :=
    Cases (MapGet ? (Fst cfg) node) of
        (NONE) => if (ad_eq node BDDzero) then (cfg,BDDone) else
                        (cfg,BDDzero)
      | (SOME (x,(l,r))) =>
         Cases bound of
```

```
            O => (* Error *) (initBDDconfig,BDDzero)
          | (S bound') =>
              (BDDmake (Fst (BDDneg_2 (Fst (BDDneg_2 cfg l bound'))
                                      r bound'))
                       x
                       (Snd (BDDneg_2 cfg l bound'))
                       (Snd (BDDneg_2 (Fst (BDDneg_2 cfg l bound'))
                                      r bound')))
      end
  end.
```

where `initBDDconfig` is the configuration containing no nodes—except for `BDDzero` and `BDDone`.

Definition 30

```
Definition initBDDstate = (M0 BDDvar*ad*ad).

Definition initBDDsharing_map = (M0 (Map (Map ad))).

Definition initBDDconfig =
                       (initBDDstate,(initBDDsharing_map,(ad_S (ad_S ad_z)))).
```

The last argument `bound` for `BDDneg_2` is the bound on the depth of the recursion required for computing negation; this is similar to the way we defined `bool_fun_of_BDD_1` recursively. The above function is slightly inefficient because of the repetition of some sub-expressions, for example `(BDDneg_2 cfg l bound')`. The standard way of eliminating this using `let` expressions (as is done in `ocaml`) does not work in Coq because of the fact in the current implementation of Coq, all let definitions `let` $x = u$ `in` v are expanded to u with all occurrences of x replaced by v at compile time, so that the same expression v is computed several times. We later show how to get rid of this problem in the definition of the function `BDDneg_1_1` which computes negation with memoization. Meanwhile, duplicating subexpressions causes no harm in proving the function correct.

For the correctness of `BDDneg_2` there are several things we need to prove. Besides proving that the node returned is interpreted as the negation of the interpretation of the node passed as argument, we also need to prove the fact that all the original nodes in the configuration are preserved (as for `BDDmake`). This is necessary because of the fact that a configuration stores not just one but several BDDs. In fact all the BDDs that a user might need to work with at any time are stored in the same configuration. In such a situation it is necessary to prove that applying some operation on some BDD does not disturb any of the BDDs already present in the configuration, including the BDD on which the operation is being applied, rather all new nodes required are added in free locations.

Note that this putting together of all BDDs in one configuration is how BDDs are intended to be used, since this allows maximum sharing of identical BDDs. Also operations like negation and disjunction require that both the BDDs are stored in the same configuration. The lemma `BDDunique` then makes it trivial to check the equivalence of two BDDs: just compare the corresponding addresses. However this requires both the BDDs to compare to be stored in the same configuration.

The lemma proved for `BDDneg_2` is as follows.

Lemma 15 *Under some standard well-formedness conditions:*

1. *the new configuration, which we obtain after computing the negation of* `node`*, is OK;*
2. *if at address* `a` *there was a node* (x, l, r) *in the old configuration (before we computed the negation), then* (x, l, r) *still is at* `a` *in the new configuration;*
3. *if* `node` *was an internal node in the old configuration, then the node* `node'` *returned by* `BDDneg_2` *on* `node` *is also internal in the new configuration, and has the same variable field;*

4. `node'` *is OK as a node in the new configuration;*

5. the semantics of `node'` *in the new configuration is the negation of that of* `node` *in the old configuration.*

In Coq:

```
Lemma BDDneg_2_lemma :
  (bound:nat)(cfg:BDDconfig)(node:ad)
  (BDDconfig_OK cfg) ->
  (config_node_OK cfg node) ->
  ( (is_internal_node cfg node) ->
    (lt (nat_of_ad (var cfg node)) bound) ) ->
  (BDDconfig_OK (Fst (BDDneg_2 cfg node bound)))
  /\ ( (x:BDDvar)(l,r,a:ad)
       (MapGet ? (Fst cfg) a)=(SOME ? (x,(l,r))) ->
       (MapGet ? (Fst (Fst (BDDneg_2 cfg node bound))) a)=
         (SOME ? (x,(l,r)))
     )
  /\ ( (is_internal_node cfg node) ->
       (is_internal_node (Fst (BDDneg_2 cfg node bound))
                         (Snd (BDDneg_2 cfg node bound)))
       /\ (var cfg node)=(var (Fst (BDDneg_2 cfg node bound))
                         (Snd (BDDneg_2 cfg node bound))) )
  /\ (config_node_OK (Fst (BDDneg_2 cfg node bound))
                     (Snd (BDDneg_2 cfg node bound)))
  /\ (bool_fun_eq (bool_fun_of_BDD (Fst (BDDneg_2 cfg node bound))
                      (Snd (BDDneg_2 cfg node bound)))
        (bool_fun_neg (bool_fun_of_BDD cfg node))).
```

Proof: Let us first define some abbreviations for the following discussion. Let `(cfgl,nodel)` be `(BDDneg_2 cfg l bound')`, the result of applying negation on node `l` in `cfg`, `(cfgr,noder)` be `(BDDneg_2 cfgl r bound')`, the result of applying negation on node `r` in `cfgl`, where `cfg`, `cfgl`, `l`, `r`, `bound'` are as used in the definition of `BDDneg_2`.

The hypotheses in the lemma are that `cfg` and `node` are OK and that `bound` is sufficiently large (which we can easily ensure as we did for `bool_fun_of_BDD`). We then prove that the new configuration and node returned are OK (items 1 and 4 in the informal description), that the original nodes are preserved (item 2), besides the fact that `BDDneg_2` indeed computes negation in terms of boolean functions (item 5).

The proof is again using well founded induction on `bound`. It is important to note that we could not prove the above properties as separate lemmas, although this would have been more practical, since indeed the proof turned out to be rather large. This is because we required all these conditions together in the induction hypothesis. For example, we first compute negation of node `l` in the configuration `cfg` to get a new configuration `cfgl`. When we apply negation on node `r`, we use this new configuration `cfgl`. For this we require that `cfgl` be OK. Also that the node `r` which was OK in the old configuration `cfg` is OK in the new configuration `cfgl`. Also the new node `nodel` is required later on for applying `BDDmake` and for using the properties of `BDDmake` we require `nodel` to be OK in the configuration `cfgl`, from which we will be able to prove that it is also OK in the next configuration `cfgr` using the induction hypothesis.

We also had to prove an additional result simultaneously, which is not essential for the correctness of negation. The property is that the node returned by negation contains the same variable as the node on which negation was applied (item 3). This condition is required because for `BDDmake` we require that the variables stored at the left and right nodes, here `nodel` and `noder`, should be less than the variable passed as argument, in this case `x`. Although this conclusion could be derived from the remaining ones in the lemma, it seemed simpler to do it the way we did it. It turned out that we had to actually prove it as a lemma later on (`bool_fun_eq_var_2`) while implementing negation with memoization, but we could not anticipate it.

Most of the conclusions follow in a straightforward way from the induction hypotheses using the properties of `BDDmake`. To prove that the configuration returned by `BDDneg_2` is OK, we use the fact that the configuration returned by `BDDmake` is OK, provided that the arguments to `BDDmake` are OK, which we have from the induction hypothesis. Similarly, we use the fact that the node returned by `BDDmake` is OK and that the original nodes in the configuration are preserved. For proving the semantics as boolean functions, we use the fact that the negation of the boolean function `(bool_fun_if x bf1 bf2)` is equal to the boolean function `(bool_fun_if x (bool_fun_neg bf1) (bool_fun_neg bf2))`. To prove that the variable stored at the node returned by negation is the same as the variable stored at the node passed as argument, we use the property of corresponding conclusion `BDDmake_semantics`. However for that, we require to show that `nodel` and `noder` are distinct. For this we require the use of `BDDunique`. If `nodel` and `noder` were the same then because of the fact that they are interpreted as respectively the negations of the interpretations of `l` and `r`, we would get that the interpretations of `l` and `r` are the same, from which, using the uniqueness lemma, we would get that `l` and `r` are same, implying that the BDD stored at `node` is not reduced, violating the hypothesis `(BDDconfig_OK cfg)`. □

4.5.2 Negation with Memoization

To implement negation with memoization, we use a map called `BDDneg_memo` to store values of negation of nodes in the configuration that have already been computed.

Definition 31

```
Definition BDDneg_memo := (Map ad).
```

A `BDDneg_memo` maps an address to another address, which corresponds to its negation.
We have the following functions for looking up and making insertions into the memoization table.

Definition 32

```
Definition BDDneg_memo_lookup :=
  [memo:BDDneg_memo; a:ad](MapGet ? memo a).
```

Definition 33

```
Definition BDDneg_memo_put :=
  [memo:BDDneg_memo; a,node:ad](MapPut ? memo a node).
```

The new negation function works by looking up the memoization table for the node whose negation is to be computed, and returns the corresponding node if it is found. Otherwise, it recursively computes the negation of the sub-BDDs and applies `BDDmake` on them. The function returns a new memoization table which has all the original nodes and the new node whose negation whose negation has just been calculated. Since the function works recursively, all the nodes in the BDD whose negation needs to be computed in the process would be added in the memoization table, unless they are already present.

Also, we may not like to discard the memoization table once the negation of the required node has been computed. This is because later on we may require to compute negation of some other BDD in the configuration, which may have nodes common with a BDD whose negation has previously been calculated, so we could use the results stored in the memoization table. So we can maintain the memoization table together with the configuration. If at some point, we do not want to use the results stored in the memoization table, then we can discard the memoization table and start with a fresh memoization table, namely the empty map `(M0 ad)`).

The following is the function to compute negation with memoization. It takes an additional argument, the memoization table and returns another memoization table.

Definition 34

```
Fixpoint BDDneg_1_1 [cfg:BDDconfig; memo:BDDneg_memo; node:ad; bound:nat]
```

```
     : (BDDconfig*ad)*BDDneg_memo :=
Cases (BDDneg_memo_lookup memo node) of
    (SOME node') => ((cfg,node'),memo)
  | (NONE) =>
 Cases (MapGet ? (Fst cfg) node) of
     NONE => if (ad_eq node BDDzero) then
                 ((cfg,BDDone),(BDDneg_memo_put memo BDDzero BDDone)) else
                 ((cfg,BDDzero),(BDDneg_memo_put memo BDDone BDDzero))
   | (SOME (x,(l,r))) =>
  Cases bound of
      O => ((initBDDconfig,BDDzero),initBDDneg_memo)
    | (S bound') =>
   Cases (BDDneg_1_1 cfg memo l bound') of
       ((cfgl,nodel),memol) =>
    Cases (BDDneg_1_1 cfgl memol r bound') of
        ((cfgr,noder),memor) =>
     Cases (BDDmake cfgr x nodel noder) of
         (cfg',node') =>
      ((cfg',node'),(BDDneg_memo_put memor node node'))
     end
    end
   end
  end
 end
end.
```

`initBDDneg_memo` is the initial memoization table containing no entries.

Definition 35

```
Definition initBDDneg_memo : BDDneg_memo := (M0 ad).
```

Notice the frequent uses of `Cases` in the function. This is to get around the peculiarity of Coq mentioned before, that expands `let` expressions while saving the module, whereas it does not expand `Cases`. Thus if we use `Cases` then the evaluation of the body of the function reduces in the expected sequential way. On the other hand, this behaviour is not carved in stone, and may change in future releases of Coq. If this happens, we shall have to resort to other tricks.

Initially we had defined negation with memoization without using `Cases`, proved its correctness, and later on proved that the two functions actually return the same value on all arguments. In this way, the correctness of `BDDneg_1_1` follows. The function without `Cases` is given below. The let expressions have been manualy expanded. Many expressions occur several times in the function.

Definition 36

```
Fixpoint BDDneg_1 [arg:((BDDconfig*ad)*BDDneg_memo); bound:nat]
       : ((BDDconfig*ad)*BDDneg_memo) :=
Cases (BDDneg_memo_lookup (Snd arg) (Snd (Fst arg))) of
    (SOME node) => (((Fst (Fst arg)),node),(Snd arg))
  | (NONE) =>
    Cases (MapGet ? (Fst (Fst (Fst arg))) (Snd (Fst arg))) of
      (NONE) => (if (ad_eq (Snd (Fst arg)) BDDzero) then
      (((Fst (Fst arg)),BDDone),(BDDneg_memo_put (Snd arg) BDDzero BDDone)) else
      (((Fst (Fst arg)),BDDzero),(BDDneg_memo_put (Snd arg) BDDone BDDzero)))
    | (SOME (x,(l,r))) =>
       Cases bound of
```

```
  O => ((initBDDconfig,BDDzero),(newMap ad))
| (S bound') =>
                (Fst (Fst (BDDneg_1
                           (((Fst (Fst (BDDneg_1
                                        (((Fst (Fst arg)),l),
                                         (Snd arg)) bound'))),
                             r),
                            (Snd (BDDneg_1
                                   (((Fst (Fst arg)),l),
                                    (Snd arg)) bound')))
                           bound'))) x
                  (Snd (Fst (BDDneg_1
                             (((Fst (Fst arg)),l),(Snd arg))
                             bound')))
                  (Snd (Fst (BDDneg_1
                             (((Fst (Fst (BDDneg_1
                                          (((Fst (Fst arg)),l),
                                           (Snd arg)) bound'))),
                               r),
                              (Snd (BDDneg_1
                                     (((Fst (Fst arg)),l),
                                      (Snd arg)) bound')))
                             bound'))))),
                (BDDneg_memo_put
                  (Snd (BDDneg_1
                         (((Fst (Fst (BDDneg_1
                                      (((Fst (Fst arg)),l),
                                       (Snd arg)) bound'))),r),
                          (Snd (BDDneg_1
                                 (((Fst (Fst arg)),l),(Snd arg))
                                 bound'))) bound'))
                  (Snd (Fst arg))
                  (Snd (BDDmake
                         (Fst (Fst (BDDneg_1
                                    (((Fst (Fst (BDDneg_1
                                                 (((Fst
                                                    (Fst arg)),
                                                   l),
                                                  (Snd arg))
                                                 bound'))),r),
                                     (Snd (BDDneg_1
                                            (((Fst (Fst arg)),l),
                                             (Snd arg)) bound')))
                                    bound'))) x
                               (Snd (Fst (BDDneg_1
                                          (((Fst (Fst arg)),l),
                                           (Snd arg)) bound')))
                               (Snd (Fst (BDDneg_1
                                          (((Fst (Fst (BDDneg_1
                                                       (((Fst
                                                          (Fst arg)),
                                                         l),
                                                        (Snd arg))
```

```
                                                bound'))),r),
                                    (Snd (BDDneg_1
                                           (((Fst (Fst arg)),l),
                                            (Snd arg)) bound')))
                                  bound')))))))

        end
    end
end.
```

Lemma 16 `BDDneg_1_1` *and* `BDDneg_1` *return equal values on all arguments.*

```
Lemma BDDneg_1_1_eq_1 :
  (bound:nat; cfg:BDDconfig; memo:BDDneg_memo; node:ad)
  (BDDneg_1_1 cfg memo node bound)=(BDDneg_1 ((cfg,node),memo) bound).
```

Thus we never execute the function `BDDneg_1`. It only helps us in proving the correctness of `BDDneg_1_1`. We define a predicate `BDDneg_memo_OK_2` which states the conditions when a memoization table for negation is OK with respect to a configuration, in order that the negation function works properly. A memoization table `negm` for negation is OK with respect to a configuration `cfg`, if for every node `n_1` mapped to another node `n_2` in the table,

- `n_1` is OK in `cfg`, and
- the negation function without memoization, when applied to `cfg` and `n_1` returns the same configuration `cfg` and the node `n_2`.

For the correctness of the above function we prove the following two facts :

- The configuration and node returned by `BDDneg_1` are the same as those returned by `BDDneg_2`. From this we will be able to prove all the properties for `BDDneg_1` that we proved for `BDDneg_2`.
- The new memoization table returned by `BDDneg_1` is OK with respect to the new configuration, assuming that the original configuration was OK. This is required so that the new memoization table can be used for further computations. The second condition in the definition of `BDDneg_memo_OK_2` is needed precisely for proving this.

We require the following lemmas about `BDDneg_2` in order to prove the correctness of `BDDneg_1`.

Lemma 17 *If* `node` *and* `node'` *are two internal nodes in a configuration* `cfg`*, such that the boolean function corresponding to* `node'` *is the negation of the boolean function corresponding to* `node` *the variables stored the two nodes are the same.*

```
Lemma bool_fun_neg_eq_var_2 :
  (cfg:BDDconfig)(node,node':ad)
  (BDDconfig_OK cfg) ->
  (is_internal_node cfg node) ->
  (is_internal_node cfg node') ->
  (bool_fun_eq (bool_fun_of_BDD cfg node')
               (bool_fun_neg (bool_fun_of_BDD cfg node))) ->
  (var cfg node)=(var cfg node').
```

Proof: This is the lemma that we could have used during the proof of `BDDneg_2`. It would have allowed us to remove one of the conclusions from the statement of `BDDneg_2_lemma` and made the proof simpler.

Instead, and since we have already proved `BDDneg_2_lemma`, we use it to prove `bool_fun_neg_eq_var_2` directly. For that we apply the function `BDDneg_2` on `cfg` and `node'` to get a new configuration `cfg1` and `node1`. According to `BDDneg_2_lemma`, `node1` will contain the same variable as `node'` and will have the same interpretation as `node`. Next we apply `BDDunique` to get the required result. □

Lemma 18 *If two nodes* `node` *and* `node'` *are present in a configuration* `cfg`*, such that the boolean function corresponding to* `node'` *is the negation of the boolean function corresponding to* `node`*, then applying the function* `BDDneg_2` *on the node* `node` *and the configuration* `cfg` *will give the same configuration* `cfg` *and node* `node'`*.*

```
Lemma BDDneg_memo_OK_1_lemma_1_1_1 :
  (bound:nat)(cfg:BDDconfig)(node,node':ad)
  (BDDconfig_OK cfg) ->
  (config_node_OK cfg node) ->
  (config_node_OK cfg node') ->
  ((is_internal_node cfg node) -> (lt (nat_of_ad (var cfg node)) bound)) ->
  (bool_fun_eq (bool_fun_of_BDD cfg node')
               (bool_fun_neg (bool_fun_of_BDD cfg node))) ->
  (BDDneg_2 cfg node bound)=(cfg,node').
```

Proof: The above lemma says that if the negation of `node` is already present in the configuration, then applying `BDDneg_2` on `node` does not add any new nodes in the configuration. The proof is by well founded induction on `bound`. For the case where `node` is an internal node (and thus `node'` is also internal from the assumptions and using `BDDunique`), we will get from induction hypothesis that applying `BDDneg_2` on the left node of `node` will give the left node of `node'` and the same configuration `cfg`, and similarly for the right node. Finally, to compute `BDDneg_2` of `node` we will require to apply `BDDmake` on `cfg` with the variable stored at `node` and the left and right nodes of `node'`. However from the previous lemma `bool_fun_neg_eq_var_2`, we conclude that the variables stored at `node` and `node'` are the same. Thus the tuple with which we want to apply `BDDmake` is already present in the configuration. This requires another property of `BDDmake`, that if a tuple is already present in a configuration, then applying `BDDmake` on the configuration with that tuple will return the same configuration and the corresponding node. □

We finally proved the following required property of `BDDneg_1`.

Lemma 19 *The configuration and node returned by* `BDDneg_1` *are the same as those returned by* `BDDneg_2`*. Also, the new memoization table returned by* `BDDneg_1` *is OK with respect to the new configuration. In Coq:*

```
Lemma BDDneg_1_lemma' : (bound:nat)(arg:((BDDconfig*ad)*BDDneg_memo))
  (BDDconfig_OK (Fst (Fst arg))) ->
  (config_node_OK (Fst (Fst arg)) (Snd (Fst arg))) ->
  (BDDneg_memo_OK_2 (Fst (Fst arg)) (Snd arg)) ->
  ((is_internal_node (Fst (Fst arg)) (Snd (Fst arg))) ->
  (lt (nat_of_ad (var (Fst (Fst arg)) (Snd (Fst arg)))) bound)) ->
  (Fst (BDDneg_1 arg bound))=(BDDneg_2 (Fst (Fst arg)) (Snd (Fst arg)) bound)
  /\ (BDDneg_memo_OK_2 (Fst (Fst (BDDneg_1 arg bound))) (Snd (BDDneg_1 arg bound))).
```

Proof: The proof is by well founded induction on `bound`. We consider the two cases in the definition of `BDDneg_1` depending on whether there is already an entry in the memoization table corresponding to `node`. If so then we use the conditions in `BDDneg_memo_OK_2` and the lemma `BDDneg_memo_OK_1_lemma_1_1_1`. □

Finally we have the following function of negation in which we have removed the last argument `bound` from `BDDneg_1_1`. It works by calling `BDDneg_1_1` with a suitable bound.

Definition 37

```
Definition BDDneg :=
  [cfg:BDDconfig; memo:BDDneg_memo; node:ad]
  Cases (BDDneg_1_1 cfg memo node (S (nat_of_ad (var cfg node)))) of
      ((cfg',node'),memo') => (cfg',(node',memo'))
  end.
```

```
BDDneg : BDDconfig -> BDDneg_memo -> ad -> BDDconfig * ad * BDDneg_memo.
```

We have the following properties of BDDneg which are straightforward applications of the previous lemmas related to BDDneg_2 and BDDneg_1_1.

Lemma 20 *The configuration returned by* BDDneg *is OK.*

```
Lemma BDDneg_keeps_config_OK
     : (cfg:BDDconfig; negm:BDDneg_memo; node:ad)
        (BDDconfig_OK cfg)
        ->(BDDneg_memo_OK_2 cfg negm)
        ->(config_node_OK cfg node)
        ->(BDDconfig_OK (Fst (BDDneg cfg negm node))).
```

Lemma 21 *The nodes that were OK in the original configuration are also OK in the new configuration returned by* BDDneg.

```
Lemma BDDneg_keeps_node_OK
     : (cfg:BDDconfig; negm:BDDneg_memo; node:ad)
        (BDDconfig_OK cfg)
        ->(BDDneg_memo_OK_2 cfg negm)
        ->(config_node_OK cfg node)
        ->(node':ad)
           (config_node_OK cfg node')
           ->(config_node_OK (Fst (BDDneg cfg negm node)) node').
```

Lemma 22 *The memoization table returned by* BDDneg *is OK with respect to the new configuration table returned.*

```
Lemma BDDneg_keeps_neg_memo_OK
     : (cfg:BDDconfig; negm:BDDneg_memo; node:ad)
        (BDDconfig_OK cfg)
        ->(BDDneg_memo_OK cfg negm)
        ->(config_node_OK cfg node)
        ->(BDDneg_memo_OK (Fst (BDDneg cfg negm node))
            (Snd (Snd (BDDneg cfg negm node)))).
```

Lemma 23 *The interpretations of all the nodes that were OK in the original configuration remain unchanged in the new configuration returned by* BDDneg.

```
Lemma BDDneg_preserves_bool_fun
     : (cfg:BDDconfig; negm:BDDneg_memo; node:ad)
        (BDDconfig_OK cfg)
        ->(BDDneg_memo_OK_2 cfg negm)
        ->(config_node_OK cfg node)
        ->(node':ad)
           (config_node_OK cfg node')
           ->(bool_fun_eq
               (bool_fun_of_BDD (Fst (BDDneg cfg negm node)) node')
               (bool_fun_of_BDD cfg node')).
```

Lemma 24 *The node returned by* BDDneg *is OK in the configuration returned.*

```
Lemma BDDneg_node_OK
     : (cfg:BDDconfig; negm:BDDneg_memo; node:ad)
        (BDDconfig_OK cfg)
```

```
        ->(BDDneg_memo_OK_2 cfg negm)
        ->(config_node_OK cfg node)
        ->(config_node_OK (Fst (BDDneg cfg negm node))
            (Fst (Snd (BDDneg cfg negm node)))).
```

Lemma 25 *The interpretation of the new node returned by* BDDneg *in the new configuration is as expected.*

```
Lemma BDDneg_is_neg
     : (cfg:BDDconfig; negm:BDDneg_memo; node:ad)
        (BDDconfig_OK cfg)
        ->(BDDneg_memo_OK_2 cfg negm)
        ->(config_node_OK cfg node)
        ->(bool_fun_eq
            (bool_fun_of_BDD (Fst (BDDneg cfg negm node))
              (Fst (Snd (BDDneg cfg negm node))))
            (bool_fun_neg (bool_fun_of_BDD cfg node))).
```

4.6 Disjunction

We directly implemented disjunction using memoization, since implementing negation first without memoization and then with memoization turned out to be somewhat complex. The memoization table for disjunction maps a pair of addresses to another address, and is curried.

Definition 38

```
Definition BDDor_memo := (Map (Map ad)).
```

The following function computes disjunction with memoization. The function is somewhat more complex than that for negation since we need to consider three cases depending upon the ordering between the variables stored at the two nodes.

The function takes a configuration, a memoization table, and two nodes as argument, and returns a node, a configuration and a memoization table.

Definition 39

```
Fixpoint BDDor_1_1 [cfg:BDDconfig; memo:BDDor_memo; node1,node2:ad; bound:nat]
   : BDDconfig*ad*BDDor_memo :=
Cases (BDDor_memo_lookup memo node1 node2) of
    (SOME node) => (cfg,(node,memo))
  | (NONE) => if (ad_eq node1 BDDzero) then
                 (cfg,(node2,(BDDor_memo_put memo BDDzero node2 node2))) else
              if (ad_eq node1 BDDone) then
                 (cfg,(BDDone,(BDDor_memo_put memo BDDone node2 BDDone))) else
              if (ad_eq node2 BDDzero) then
                 (cfg,(node1,(BDDor_memo_put memo node1 BDDzero node1))) else
              if (ad_eq node2 BDDone) then
                 (cfg,(BDDone,(BDDor_memo_put memo node1 BDDone BDDone))) else
 Cases bound of
     O => (initBDDconfig,(BDDzero,initBDDor_memo))
   | (S bound') =>
  Cases (BDDcompare (var cfg node1) (var cfg node2)) of
      EGAL =>
   Cases (BDDor_1_1 cfg memo (low cfg node1) (low cfg node2) bound') of
       (cfgl,(nodel,memol)) =>
    Cases (BDDor_1_1 cfgl memol (high cfg node1) (high cfg node2) bound') of
```

```
        (cfgr,(noder,memor)) =>
     Cases (BDDmake cfgr (var cfg node1) nodel noder) of
         (cfg',node') =>
           (cfg',(node',(BDDor_memo_put memor node1 node2 node')))
     end
    end
   end
    | INFERIEUR =>
   Cases (BDDor_1_1 cfg memo node1 (low cfg node2) bound') of
       (cfgl,(nodel,memol)) =>
    Cases (BDDor_1_1 cfgl memol node1 (high cfg node2) bound') of
        (cfgr,(noder,memor)) =>
     Cases (BDDmake cfgr (var cfg node2) nodel noder) of
         (cfg',node') =>
           (cfg',(node',(BDDor_memo_put memor node1 node2 node')))
     end
    end
   end
    | SUPERIEUR =>
   Cases (BDDor_1_1 cfg memo (low cfg node1) node2 bound') of
       (cfgl,(nodel,memol)) =>
    Cases (BDDor_1_1 cfgl memol (high cfg node1) node2 bound') of
        (cfgr,(noder,memor)) =>
     Cases (BDDmake cfgr (var cfg node1) nodel noder) of
         (cfg',node') =>
           (cfg',(node',(BDDor_memo_put memor node1 node2 node')))
     end
    end
   end
  end
 end
end.

Definition BDDor :=
  [cfg:BDDconfig; memo:BDDor_memo; node1,node2:ad]
  (BDDor_1_1 cfg memo node1 node2
             (S (max (nat_of_ad (var cfg node1)) (nat_of_ad (var cfg node2))))).

BDDor : BDDconfig->BDDor_memo->ad->ad->BDDconfig*ad*BDDor_memo
```

The functions `BDDor_memo_lookup` and `BDDor_memo_put` are used for respectively looking up and making insertions into a memoization table.

Definition 40

```
Definition BDDor_memo_lookup := [memo:BDDor_memo; node1,node2:ad]
  Cases (MapGet ? memo node1) of
      (NONE) => (NONE ad)
    | (SOME m1) =>
        Cases (MapGet ? m1 node2) of
            (NONE) => (NONE ad)
          | (SOME node) => (SOME ad node)
        end
  end.
```

Definition 41

```
Definition BDDor_memo_put := [memo:BDDor_memo; node1,node2,node:ad]
  let m1 = Cases (MapGet ? memo node1) of
               (SOME y) => y
             | (NONE) => (newMap ad)
           end in
  let m1' = (MapPut ? m1 node2 node) in
     (MapPut ? memo node1 m1').
```

`initBDDor_memo` is the initial memoization table for disjunction containing no entries.

Definition 42

```
Definition initBDDor_memo := (M0 (Map ad)).
```

As with negation, we had initially defined a function without `Cases` which contained many repetitions of expressions. Its definition runs over 4.5 A4 pages, and is not particulary illuminating, so we shall leave it implicit.

Definition 43

```
Fixpoint BDDor_1 [cfg:BDDconfig; memo:BDDor_memo; node1,node2:ad; bound:nat]
   : BDDconfig*ad*BDDor_memo :=
Cases (BDDor_memo_lookup memo node1 node2) of
    (SOME node) => (cfg,(node,memo))
  | (NONE) =>
    ...
end.
```

The equality of the two functions `BDDor_1_1` and `BDDor_1` is stated in the following lemma.

Lemma 26

```
Lemma BDDor_1_1_eq_1 :
  (bound:nat; cfg:BDDconfig; memo:BDDor_memo; node1,node2:ad)
  (BDDor_1_1 cfg memo node1 node2 bound)=(BDDor_1 cfg memo node1 node2 bound).
```

`BDDor_1` when applied to a configuration `cfg`, memoization table `memo` and nodes `node1` and `node2` works as follows. We describe the case where both the nodes are internal. In the case where both the nodes contain the same variable, we first recursively compute the disjunction of the left nodes of `node1` and `node2` with the configuration `cfg` and memoization table `memo` to get a new configuration `cfgl` and memoization table `meml` and a node `nodel`. We now compute the disjunction of the two right nodes of `node1` and `node2` with the new configuration `cfgl` and memoization table `meml` to get a configuration `cfgr`, memoization table `memor` and node `noder`. We now apply `BDDmake` to the nodes `nodel` and `noder` and the variable stored at `node1` in the configuration `cfgr` to get the final configuration and the required node. We make a corresponding entry in the memoization table `memor` to get the final memoization table (`memor` already contains entries corresponding to the disjunction of intermediate nodes computed by the recursive calls). The function follows from the orthogonality of disjunction (see Section 2). The use of `BDDmake` takes care of the cases where `nodel` and `noder` may be equal, and also the case where the required node may be already present in the configuration. Also for applying `BDDmake` we need to ensure that the variable stored at `nodel` and `noder` are less than the variable stored at `node`. So we put this extra conclusion in the lemma `BDDor_1_lemma` described below.

In the case where the variable stored at `node1` is greater that the variable stored at `node2`, we compute disjunction differently which corresponds to the following property of disjunction.

$$(A \longrightarrow F_1; G_1) \vee F_2 =_{\mathrm{df}} A \longrightarrow (F_1 \vee F_2); (G_1 \vee F_2)$$

In this case we do not need to consider the sons of `node2`. The case where the variable at `node2` is less than the variable at `node2` is also similar.

We have the predicate `BDDor_memo_OK` which describes the conditions when a memoization table for disjunction is OK with respect to a configuration.

Definition 44 *A memoization table* `orm` *is said to be OK with respect to a configuration* `cfg` *if for any pair of nodes* (n_1, n_2) *which is mapped to a node* n_3 *in the table* `orm`*,*

1. n_1*,* n_2 *and* n_3 *are OK in the configuration* `cfg`*,*
2. *the interpretation of* n_3 *as boolean function is the disjunction of the interpretations of* n_1 *and* n_2 *as boolean functions in* `cfg`*.*
3. *the variable stored at* n_3 *is less than or equal to the variable stored at* n_1 *or to that stored at* n_2*.*

In Coq:

```
Definition BDDor_memo_OK := [cfg:BDDconfig; memo:BDDor_memo]
  (node1,node2,node:ad)
  (BDDor_memo_lookup memo node1 node2)=(SOME ? node) ->
  (config_node_OK cfg node1)
  /\ (config_node_OK cfg node2)
  /\ (config_node_OK cfg node)
  /\ (ad_le (var cfg node)
            (BDDvar_max (var cfg node1) (var cfg node2)))=true
  /\ (bool_fun_eq (bool_fun_of_BDD cfg node)
                  (bool_fun_or (bool_fun_of_BDD cfg node1)
                               (bool_fun_of_BDD cfg node2))).
```

where `BDDvar_max` takes the maximum of two BDD variables.

The third condition above could actually be proved from the previous two provided the configuration `cfg` was OK. However proving the correctness of the function `BDDor_1` using the above definition of `BDDor_memo_OK` does not imvolve any additional complexity. Also since we prove that the memoization table returned by `BDDor_1` also satisfies all these conditions, we can easily use the new memoization table for further computations. This condition is required for proving the lemma `BDDor_1_lemma` described below, where we need to prove that applying disjunction on two nodes returns a node which does not contain a variable greater than the maximum of the variables stored at the two nodes to be able to apply the induction hypothesis.

Lemma 27 *The* `BDDor_1` *function is correct:*

```
Lemma BDDor_1_lemma :
  (bound:nat; cfg:BDDconfig; node1,node2:ad; memo:BDDor_memo)
  (BDDconfig_OK cfg) ->
  (BDDor_memo_OK cfg memo) ->
  (config_node_OK cfg node1) ->
  (config_node_OK cfg node2) ->
  ((is_internal_node cfg node1) -> (is_internal_node cfg node2) ->
   (lt (max (nat_of_ad (var cfg node1)) (nat_of_ad (var cfg node2))) bound)) ->
  (BDDconfig_OK (Fst (BDDor_1 cfg memo node1 node2 bound)))
  /\ (BDDor_memo_OK (Fst (BDDor_1 cfg memo node1 node2 bound))
                    (Snd (Snd (BDDor_1 cfg memo node1 node2 bound))))
  /\ (config_node_OK (Fst (BDDor_1 cfg memo node1 node2 bound))
                     (Fst (Snd (BDDor_1 cfg memo node1 node2 bound))))
  /\ (nodes_preserved cfg (Fst (BDDor_1 cfg memo node1 node2 bound)))
  /\ (BDDvar_le (var (Fst (BDDor_1 cfg memo node1 node2 bound))
```

```
                    (Fst (Snd (BDDor_1 cfg memo node1 node2 bound))))
               (BDDvar_max (var cfg node1) (var cfg node2)))=true
  /\ (bool_fun_eq (bool_fun_of_BDD
                    (Fst (BDDor_1 cfg memo node1 node2 bound))
                    (Fst (Snd (BDDor_1 cfg memo node1 node2 bound))))
                  (bool_fun_or (bool_fun_of_BDD cfg node1)
                               (bool_fun_of_BDD cfg node2))).
```

Proof: The proof uses well founded induction on `bound`.

The hypotheses on the arguments are the usual ones, namely that the configuration `cfg` and memoization table `memo` are OK and the two nodes `node1` and `node2` are OK. We require to prove the usual conclusions that the configuration, memoization table and node returned are OK. We also prove that the nodes already present in `cfg` are not disturbed. Finally we have the main property of `BDDor_1` which states that the interpretations of the nodes in the argument and the resulting node are related in the expected way: the result, as a boolean function, is the disjunction of the boolean functions denoted by the arguments.

As mentioned before we also have the additional property that the variable of the node returned is not greater than the maximum of the variables of the two nodes in the argument. This is required because when we apply `BDDmake` on the nodes `nodel` and `noder` with the variable of `node1` (in the case where both the nodes are internal and contain the same variable) we need to be sure that the variable stored at `node1` is strictly greater than the variables stored at `nodel` and `noder`. For this we will have in the induction hypotheses that if `nodel` contains a variable, then it is not greater than the maximum of the variables of the left nodes of `node1` and `node2`. Now to prove that the variable of `nodel` is strictly less than the variable of `node1`, we use the fact that the variables on any path in the BDDs are stored in strictly decreasing order. So the variable stored at the left node of `node1` will be strictly less than the variable stored at `node1` (using the lemma `BDDvar_ordered_high`). Similarly the variable at the left node of `node2` will be strictly less than the variable stored at `node2` (which is equal to the variable stored at `node1` in this case). Since the variable stored at `nodel` is not greater than the maximum of the variables stored at the left nodes of `node1` and `node2`, we conclude that the variable stored at `nodel` is strictly less than the variable stored at `node1`. However the argument is actually more complex than this as discussed below.

Note in the definition of the function `var`, which returns the variable stored at a node in a configuration, that we need to consider the case where the node is a leaf node, in which case there is no variable stored at the node. Since the function needs to be total, in our definition, we arbitrarily chose to return the value `ad_z` (the zeroth variable) in case the node is a leaf node. The other alternative, of having the function return results like `NONE` or `(SOME x)`, would have increased the length of the statements of lemmas like `BDDor_1_lemma` by several times.

However the problem with our definition of `var` is in the ordering property of `BDDs`. Note in the statement of the lemma `BDDvar_ordered_high` (and similarly in `BDDvar_ordered_low`), which states that the variable stored at the right node of a node `node` is strictly less than the variable stored at `node`, we require the condition that the right node be an internal node. In case the right node is a leaf node, then the function `var` would return the zeroth variable. But it could be possible that `node` also contains the same variable, in which case `var` of the right node would not be less that the var of `node`. (We discuss other ways out later on.)

Returning to the proof of the `BDDor_1_lemma`, we cannot use the argument that the left node of `node1` contains a variable less than the variable contained by `node1`, and similarly for `node2`. This is because the left node could be a leaf node. We get around this problem by proving the following lemmas.

Lemma 28

```
Lemma BDDvar_ordered_low_1 : (cfg:BDDconfig; node1,node2:ad)
  (BDDconfig_OK cfg) ->
  (is_internal_node cfg node1) ->
  (is_internal_node cfg node2) ->
  ((is_internal_node cfg (low cfg node1))
    \/ (is_internal_node cfg (low cfg node2))) ->
```

```
  (BDDcompare (BDDvar_max (var cfg (low cfg node1))
                          (var cfg (low cfg node2)))
              (BDDvar_max (var cfg node1) (var cfg node2))) =INFERIEUR.
```

In the above lemma, we do not require the left nodes of both `node1` and `node2` to be internal nodes, only one of them needs to be. The final conclusion is all we need for our purposes. But we first need to ensure that at least one of the left nodes is internal. This we get from the following lemma.

Lemma 29 *If the disjunction of* `node1` *and* `node2` *is an internal node then at least one of* `node1` *and* `node2` *is an internal node.*

```
Lemma BDDor_1_internal :
  (cfg:BDDconfig)(memo:BDDor_memo)(node1,node2:ad)(bound:nat)
  (BDDconfig_OK cfg) ->
  (config_node_OK cfg node1) ->
  (config_node_OK cfg node2) ->
  (BDDor_memo_OK cfg memo) ->
  (is_internal_node (Fst (BDDor_1 cfg memo node1 node2 bound))
                    (Fst (Snd (BDDor_1 cfg memo node1 node2 bound)))) ->
  ((is_internal_node cfg node1) \/ (is_internal_node cfg node2)).
```

Thus in the proof of `BDDor_1_lemma` we can use the argument that if `node1` is an internal node, then by `BDDor_1_internal`, among the left nodes of `node1` and `node2`, at least one will be an internal node. Then from `BDDvar_ordered_low_1` we conclude that the maximum of the variables of the left nodes of `node1` and `node2` is strictly less than the variable stored at `node1` (which is equal to the variable stored at `node2`).

`BDDvar_ordered_low_1` is also required for being able to apply induction hypotheses in the first place. For being able to apply induction hypotheses first we need to prove that the predecessor of `bound` is strictly greater than the maximum of the variables of the left nodes of `node1` and `node2`.

Similarly we require a lemma for the right nodes which is stated below.

Lemma 30

```
Lemma BDDvar_ordered_high_1 : (cfg:BDDconfig; node1,node2:ad)
  (BDDconfig_OK cfg) ->
  (is_internal_node cfg node1) ->
  (is_internal_node cfg node2) ->
  ((is_internal_node cfg (high cfg node1))
    \/ (is_internal_node cfg (high cfg node2))) ->
  (BDDcompare (BDDvar_max (var cfg (high cfg node1))
                          (var cfg (high cfg node2)))
              (BDDvar_max (var cfg node1) (var cfg node2))) =INFERIEUR.
```

The above discussion was for the case where the variables stored at the left and right nodes are equal. The arguments for the remaining two cases are slightly simpler. For example, in the case where the variable stored at `node1` is greater than the variable stored at `node2`, we first take the disjunction of the left node of `node1` with `node2`. In this case, the variable stored at the left node of `node1` will be less than the variable stored at `node1` irrespective of whether the left node is an internal node or not. This is because even if the left node is a leaf node (in which case `var` of the left node will be the zeroth variable), we are assured the variable of `node1` will not be the zeroth variable because we already have the condition that the variable stored at `node2` is less than the variable stored at `node1` and there is no variable less than the zeroth variable. In this way, we can easily prove the following lemmas.

Lemma 31

```
Lemma BDDvar_ordered_low_2 : (cfg:BDDconfig; node1,node2:ad)
```

```
  (BDDconfig_OK cfg) ->
  (is_internal_node cfg node1) ->
  (is_internal_node cfg node2) ->
  (BDDcompare (var cfg node1) (var cfg node2))=INFERIEUR ->
  (BDDcompare (BDDvar_max (var cfg node1) (var cfg (low cfg node2)))
              (var cfg node2))=INFERIEUR.
```

Lemma 32

```
Lemma BDDvar_ordered_high_2 : (cfg:BDDconfig; node1,node2:ad)
  (BDDconfig_OK cfg) ->
  (is_internal_node cfg node1) ->
  (is_internal_node cfg node2) ->
  (BDDcompare (var cfg node1) (var cfg node2))=INFERIEUR ->
  (BDDcompare (BDDvar_max (var cfg node1) (var cfg (high cfg node2)))
              (var cfg node2))=INFERIEUR.
```

Lemma 33

```
Lemma BDDvar_ordered_low_3 : (cfg:BDDconfig; node1,node2:ad)
  (BDDconfig_OK cfg) ->
  (is_internal_node cfg node1) ->
  (is_internal_node cfg node2) ->
  (BDDcompare (var cfg node2) (var cfg node1))=INFERIEUR ->
  (BDDcompare (BDDvar_max (var cfg (low cfg node1)) (var cfg node2))
              (var cfg node1))=INFERIEUR.
```

Lemma 34

```
Lemma BDDvar_ordered_high_3 : (cfg:BDDconfig; node1,node2:ad)
  (BDDconfig_OK cfg) ->
  (is_internal_node cfg node1) ->
  (is_internal_node cfg node2) ->
  (BDDcompare (var cfg node2) (var cfg node1))=INFERIEUR ->
  (BDDcompare (BDDvar_max (var cfg (high cfg node1)) (var cfg node2))
              (var cfg node1))=INFERIEUR.
```

Finally, we must mention that one of the reasons why the proof was so long was the problem related to the definition of `var`. In fact this problem could be avoided if the variables in the BDDs started from one onwards instead of zero onwards. However we could also have avoided the problem by using a different quantity for applying induction instead of the variable stored at the node. For example, we could use a quantity `(f node)` where f would be defined as `(f BDDzero) = (f BDDone) = ad_z` and `(f internal_node) = 1 + (var node)`. This quantity strictly decreases along any path even if it involves leaf ndes. This problem was there in all the proofs, however it posed the maximum difficulty in the proof of disjunction. Since this problem was not foreseen in the beginning we decided to stick to the present method. □

The final function for disjunction, in which the extra argument `bound` is not required is now defined.

Definition 45

```
Definition BDDor :=
  [cfg:BDDconfig; memo:BDDor_memo; node1,node2:ad]
  (BDDor_1_1 cfg memo node1 node2
             (S (max (nat_of_ad (var cfg node1)) (nat_of_ad (var cfg node2))))).
```

We proved the folowing propperties of BDDor. They are straightforward applications of the previous lemmas.

Lemma 35 *The configuration returned by* BDDor *is OK.*

```
Lemma BDDor_keeps_config_OK
     : (cfg:BDDconfig; orm:BDDor_memo; node1,node2:ad)
        (BDDconfig_OK cfg)
        ->(BDDor_memo_OK cfg orm)
        ->(config_node_OK cfg node1)
        ->(config_node_OK cfg node2)
        ->(BDDconfig_OK (Fst (BDDor cfg orm node1 node2))).
```

Lemma 36 BDDor *preserves the original nodes in the configuration.*

```
Lemma BDDor_keeps_node_OK
     : (cfg:BDDconfig; orm:BDDor_memo; node1,node2:ad)
        (BDDconfig_OK cfg)
        ->(BDDor_memo_OK cfg orm)
        ->(config_node_OK cfg node1)
        ->(config_node_OK cfg node2)
        ->(node:ad)
           (config_node_OK cfg node)
           ->(config_node_OK (Fst (BDDor cfg orm node1 node2)) node).
```

Lemma 37 *The memoization table returned by* BDDor *is OK.*

```
Lemma BDDor_keeps_or_memo_OK
     : (cfg:BDDconfig; orm:BDDor_memo; node1,node2:ad)
        (BDDconfig_OK cfg)
        ->(BDDor_memo_OK cfg orm)
        ->(config_node_OK cfg node1)
        ->(config_node_OK cfg node2)
        ->(BDDor_memo_OK (Fst (BDDor cfg orm node1 node2))
            (Snd (Snd (BDDor cfg orm node1 node2)))).
```

Lemma 38 BDDor *preserves the semantics of the original nodes in the configuration.*

```
Lemma BDDor_preserves_bool_fun
     : (cfg:BDDconfig; orm:BDDor_memo; node1,node2:ad)
        (BDDconfig_OK cfg)
        ->(BDDor_memo_OK cfg orm)
        ->(config_node_OK cfg node1)
        ->(config_node_OK cfg node2)
        ->(node:ad)
           (config_node_OK cfg node)
           ->(bool_fun_eq
               (bool_fun_of_BDD (Fst (BDDor cfg orm node1 node2)) node)
               (bool_fun_of_BDD cfg node)).
```

Lemma 39 *The node returned by* BDDor *is OK.*

```
Lemma BDDor_node_OK
     : (cfg:BDDconfig; orm:BDDor_memo; node1,node2:ad)
        (BDDconfig_OK cfg)
        ->(BDDor_memo_OK cfg orm)
```

```
        ->(config_node_OK cfg node1)
        ->(config_node_OK cfg node2)
        ->(config_node_OK (Fst (BDDor cfg orm node1 node2))
            (Fst (Snd (BDDor cfg orm node1 node2)))).
```

Lemma 40 *The interpretations of the nodes in the argument and the node returned by* `BDDor` *are related in the expected way.*

```
Lemma BDDor_is_or
     : (cfg:BDDconfig; orm:BDDor_memo; node1,node2:ad)
        (BDDconfig_OK cfg)
        ->(BDDor_memo_OK cfg orm)
        ->(config_node_OK cfg node1)
        ->(config_node_OK cfg node2)
        ->(bool_fun_eq
            (bool_fun_of_BDD (Fst (BDDor cfg orm node1 node2))
              (Fst (Snd (BDDor cfg orm node1 node2))))
            (bool_fun_or (bool_fun_of_BDD cfg node1)
              (bool_fun_of_BDD cfg node2))).
```

4.7 Other Operations

The other operations on the BDDs (conjunction, implication, double implication) were implemented using negation and disjunction. For example to compute conjunction of two nodes, we successively call negation twice, followed by disjunction and negation. At each step we update the memoization tables for negation and disjunction. We do not have separate memoization tables for these functions as it would have led to too much wastage of space, since we keep all these tables for further computations. By implementing all the remaining functions using negation and disjunction, we are able to use the respective memoization tables.

Definition 46

```
Definition BDDand :=
  [cfg:BDDconfig; negm:BDDneg_memo; orm:BDDor_memo; node1,node2:ad]
  Cases (BDDneg cfg negm node1) of
      (cfg',(node1',negm')) =>
   Cases (BDDneg cfg' negm' node2) of
       (cfg'',(node2',negm'')) =>
    Cases (BDDor cfg'' orm node1' node2') of
        (cfg''',(node,orm')) =>
     Cases (BDDneg cfg''' negm'' node) of
         (cfg'''',(node',negm''')) => (cfg'''',(node',(negm''',orm')))
     end
    end
   end
  end.

BDDand  : BDDconfig -> BDDneg_memo -> BDDor_memo -> ad -> ad
          -> BDDconfig * ad * BDDneg_memo * BDDor_memo.
```

Definition 47

```
Definition BDDimpl :=
  [cfg:BDDconfig; negm:BDDneg_memo; orm:BDDor_memo; node1,node2:ad]
  Cases (BDDneg cfg negm node1) of
      (cfg',(node1',negm')) =>
```

```
   Cases (BDDor cfg' orm node1' node2) of
       (cfg'',(node,orm')) => (cfg'',(node,(negm',orm')))
   end
  end.

BDDimpl : BDDconfig -> BDDneg_memo -> BDDor_memo -> ad -> ad
          -> BDDconfig * ad * BDDneg_memo * BDDor_memo.
```

Definition 48

```
Definition BDDiff :=
  [cfg:BDDconfig; negm:BDDneg_memo; orm:BDDor_memo; node1,node2:ad]
  Cases (BDDimpl cfg negm orm node1 node2) of
      (cfg',(node1',(negm',orm'))) =>
   Cases (BDDimpl cfg' negm' orm' node2 node1) of
       (cfg'',(node2',(negm'',orm''))) =>
            (BDDand cfg'' negm'' orm'' node1' node2')
   end
  end.

BDDiff  : BDDconfig -> BDDneg_memo -> BDDor_memo -> ad -> ad
          -> BDDconfig * ad * BDDneg_memo * BDDor_memo.
```

To prove the correctness of these functions, we had to prove additional properties of negation and disjunction. Their requirement can be illustrated from the following example.

Suppose at some point, we have with us a configuration `cfg`, a memoization table for negation `negm` and a memoization table for disjunction `orm`. In case these were obtained by using the operations described, then we should have the property that `negm` is OK with respect to `cfg`, and similarly `orm` is OK with respect to `cfg`. Now suppose we apply negation on `cfg`. We would supply the argument `negm`. As a result, we would get a new configuration `cfg'` and a new memoization table `negm'`. From the properties proved for negation, we would be able to prove that `negm'` is OK with respect to `cfg'`, and hence can be used for further computations. However, if we want to use `orm` for further computations, we need to prove that `orm` is OK with respect to the new configuration. Contrarily to intuition, this is not trivial: the point is that the configuration *changed*, and that altough `orm` did not, the fact that `orm` is still OK depends heavily on the configuration.

We require to prove a similar property for disjunction. These can be derived from the following more general results.

Lemma 41 *If the nodes of* `cfg` *are preserved in* `cfg'` *and* `orm` *is OK with respect to* `cfg`*, then* `orm` *is also OK with respect to* `cfg'`*.*

```
Lemma nodes_preserved_orm_OK :
  (cfg,cfg':BDDconfig; orm:BDDor_memo)
  (BDDconfig_OK cfg) ->
  (BDDconfig_OK cfg') ->
  (nodes_preserved cfg cfg') ->
  (BDDor_memo_OK cfg orm) ->
  (BDDor_memo_OK cfg' orm).
```

Lemma 42 *If the nodes of* `cfg` *are preserved in* `cfg'` *and* `negm` *is OK with respect to* `cfg`*, then* `negm` *is also OK with respect to* `cfg'`*.*

```
Lemma nodes_preserved_negm_OK :
  (cfg,cfg':BDDconfig; negm:BDDneg_memo)
  (BDDconfig_OK cfg) ->
```

```
  (BDDconfig_OK cfg') ->
  (nodes_preserved cfg cfg') ->
  (BDDneg_memo_OK_2 cfg negm) ->
  (BDDneg_memo_OK_2 cfg' negm).
```

We have already proved that negation and disjunction preserve the nodes in the configuration. Thus we can apply the above two lemmas to prove the following results.

Lemma 43

```
Lemma BDDneg_keeps_or_memo_OK
     : (cfg:BDDconfig; negm:BDDneg_memo; orm:BDDor_memo; node:ad)
        (BDDconfig_OK cfg)
        ->(BDDneg_memo_OK cfg negm)
        ->(config_node_OK cfg node)
        ->(BDDor_memo_OK cfg orm)
        ->(BDDor_memo_OK (Fst (BDDneg cfg negm node)) orm).
```

Lemma 44

```
Lemma BDDor_keeps_neg_memo_OK
     : (cfg:BDDconfig; negm:BDDneg_memo; orm:BDDor_memo; node1,node2:ad)
        (BDDconfig_OK cfg)
        ->(BDDor_memo_OK cfg orm)
        ->(config_node_OK cfg node1)
        ->(config_node_OK cfg node2)
        ->(BDDneg_memo_OK cfg negm)
        ->(BDDneg_memo_OK (Fst (BDDor cfg orm node1 node2)) negm)).
```

The properties for `BDDand`, `BDDimpl` and `BDDiff` are similar to those for negation and disjunction. We proved the following properties for each of them by direct application of the properties of negation and disjunction.

1. The new configuration returned is OK;
2. The new node returned is OK in the new configuration;
3. The original nodes in the configuration are preserved, i.e., their interpretations remain the same;
4. The new memoization tables for disjunction and negation returned are OK with respect to the new configuration.

As to the latter, note that all these functions require both the memoization tables as argument, unlike the functions for negation and disjunction, which only needed one.

4.8 Translating propositional formulae

Finally we implemented a function to build a BDD from a propositional formula. For this we used a function `BDDvar_make` to build a BDD for a single variable. This was a direct application of `BDDmake`.

Definition 49

```
Definition BDDvar_make := [cfg:BDDconfig; x:BDDvar]
  (BDDmake cfg x BDDzero BDDone).

BDDvar_make : BDDconfig->BDDvar->BDDconfig*ad.
```

We have the following corresponding operation on boolean function.

Definition 50

```
Definition bool_fun_var : BDDvar -> bool_fun :=
  [x:BDDvar]([vb:var_binding](vb x)).
```

`(bool_fun_var x)` represents the boolean expression `x`. It is true in an environment if and only if the environment binds the variable `x` to true.

We have the following property of `BDD_var_make`.

Lemma 45

```
Lemma BDDvar_make_is_var :
  (cfg:BDDconfig; x:BDDvar)
  (BDDconfig_OK cfg) ->
  (bool_fun_eq (bool_fun_of_BDD (Fst (BDDvar_make cfg x))
                                (Snd (BDDvar_make cfg x)))
               (bool_fun_var x)).
```

Besides that, we also require to prove the usual properties for `BDDvar_make`, that the configuration and node returned are OK, the memoization tables remain OK with respect to the new configuration (`BDDvar_make` does not change memoization tables), and the original nodes are preserved. We invoke the corresponding properties for `BDDmake`.

We have the following natural definition of boolean expressions.

Definition 51

```
Inductive bool_expr : Set :=
      Zero : bool_expr
    | One : bool_expr
    | Var : BDDvar->bool_expr
    | Neg : bool_expr->bool_expr
    | Or : bool_expr->bool_expr->bool_expr
    | And : bool_expr->bool_expr->bool_expr
    | Impl : bool_expr->bool_expr->bool_expr
    | Iff : bool_expr->bool_expr->bool_expr.
```

Boolean expressions are interpreted as boolean functions in the natural way as defined by the following function.

Definition 52

```
Fixpoint bool_fun_of_bool_expr [be:bool_expr] : bool_fun :=
  Cases be of
      Zero => bool_fun_zero
    | One => bool_fun_one
    | (Var x) => (bool_fun_var x)
    | (Neg be') => (bool_fun_neg (bool_fun_of_bool_expr be'))
    | (Or be1 be2) => (bool_fun_or (bool_fun_of_bool_expr be1)
                                   (bool_fun_of_bool_expr be2))
    | (And be1 be2) => (bool_fun_and (bool_fun_of_bool_expr be1)
                                   (bool_fun_of_bool_expr be2))
    | (Impl be1 be2) => (bool_fun_impl (bool_fun_of_bool_expr be1)
                                   (bool_fun_of_bool_expr be2))
    | (Iff be1 be2) => (bool_fun_iff (bool_fun_of_bool_expr be1)
                                   (bool_fun_of_bool_expr be2))
  end.
```

Finally, we have the following function to construct a BDD from a boolean expression. This is defined in the obvious way using the functions for negation, disjunction, comjunction, etc. The functions take the two memoization tables as arguments. We take care to keep adding the results of all the intermediate computations in the memoization tables and using them.

Definition 53

```
Fixpoint BDDof_bool_expr
  [cfg:BDDconfig; negm:BDDneg_memo; orm:BDDor_memo; be:bool_expr]
  : BDDconfig*ad*BDDneg_memo*BDDor_memo :=
  Cases be of
      Zero => (cfg,(BDDzero,(negm,orm)))
    | One => (cfg,(BDDone,(negm,orm)))
    | (Var x) => Cases (BDDvar_make cfg x) of
                   (cfg',node) => (cfg',(node,(negm,orm)))
                 end
    | (Neg be') => Cases (BDDof_bool_expr cfg negm orm  be') of
                     (cfg',(node,(negm',orm'))) =>
                   Cases (BDDneg cfg' negm' node) of
                       (cfg'',(node',negm'')) => (cfg'',(node',(negm'',orm')))
                   end
                  end
    | (Or be1 be2) => Cases (BDDof_bool_expr cfg negm orm be1) of
                        (cfg',(node1,(negm',orm'))) =>
                      Cases (BDDof_bool_expr cfg' negm' orm' be2) of
                          (cfg'',(node2,(negm'',orm''))) =>
                       Cases (BDDor cfg'' orm'' node1 node2) of
                           (cfg''',(node,orm''')) =>
                                 (cfg''',(node,(negm'',orm''')))
                       end
                      end
                     end
    | (And be1 be2) => Cases (BDDof_bool_expr cfg negm orm be1) of
                         (cfg',(node1,(negm',orm'))) =>
                       Cases (BDDof_bool_expr cfg' negm' orm' be2) of
                           (cfg'',(node2,(negm'',orm''))) =>
                                   (BDDand cfg'' negm'' orm'' node1 node2)
                       end
                      end
    | (Impl be1 be2) => Cases (BDDof_bool_expr cfg negm orm be1) of
                         (cfg',(node1,(negm',orm'))) =>
                       Cases (BDDof_bool_expr cfg' negm' orm' be2) of
                           (cfg'',(node2,(negm'',orm''))) =>
                                     (BDDimpl cfg'' negm'' orm'' node1 node2)
                       end
                      end
    | (Iff be1 be2) => Cases (BDDof_bool_expr cfg negm orm be1) of
                         (cfg',(node1,(negm',orm'))) =>
                       Cases (BDDof_bool_expr cfg' negm' orm' be2) of
                           (cfg'',(node2,(negm'',orm''))) =>
                                   (BDDiff cfg'' negm'' orm'' node1 node2)
                       end
                      end
  end.
```

```
BDDof_bool_expr : BDDconfig ->BDDneg_memo ->BDDor_memo ->bool_expr
                  ->BDDconfig*ad*BDDneg_memo*BDDor_memo
```

The following lemma states the corectness of the above function. The proof is straightforward and uses the lemmas for the logical operations proved so far.

Lemma 46

```
Lemma BDDof_bool_expr_correct :
  (be:bool_expr; cfg:BDDconfig; negm:BDDneg_memo; orm:BDDor_memo)
  (BDDconfig_OK cfg) ->
  (BDDneg_memo_OK cfg negm) ->
  (BDDor_memo_OK cfg orm) ->
  (BDDconfig_OK (Fst (BDDof_bool_expr cfg negm orm be)))
  /\ (config_node_OK (Fst (BDDof_bool_expr cfg negm orm be))
                     (Fst (Snd (BDDof_bool_expr cfg negm orm be))))
  /\ (BDDneg_memo_OK (Fst (BDDof_bool_expr cfg negm orm be))
                     (Fst (Snd (Snd (BDDof_bool_expr cfg negm orm be)))))
  /\ (BDDor_memo_OK (Fst (BDDof_bool_expr cfg negm orm be))
                    (Snd (Snd (Snd (BDDof_bool_expr cfg negm orm be)))))
  /\ (nodes_preserved cfg (Fst (BDDof_bool_expr cfg negm orm be)))
  /\ (bool_fun_eq (bool_fun_of_BDD (Fst (BDDof_bool_expr cfg negm orm be))
                                   (Fst(Snd (BDDof_bool_expr cfg negm orm be))))
                  (bool_fun_of_bool_expr be)).
```

Using the above functions, it is possible to check validity, satisfiability of a formula, check equivalence of two formulas, etc. For example we defined the following function to check whether a booean expression is a tautology. For this we just build up the BDD starting with the empty configuration and memoization tables (We could use any other configuration table and memoization tables as long as they are OK). Then we just check whether the node returned is the leaf node `BDDone`.

Definition 54

```
Definition is_tauto := [be:bool_expr]
  (ad_eq BDDone (Fst (Snd (BDDof_bool_expr initBDDconfig initBDDneg_memo
                                           initBDDor_memo be)))).
```

The following lemma states the correctness of the above function. It follows immediately from the uniqueness property of BDDs and the correctness of `BDDof_bool_expr` proved above.

Lemma 47 `is_tauto` *returns true for a boolean expression if and only if the boolean expression is a tautology.*

```
Lemma is_tauto_lemma : (be:bool_expr)
  (bool_fun_eq (bool_fun_of_bool_expr be) bool_fun_one)
  <-> (is_tauto be)=true.
```

We require the fact that the initial configuration and memoization tables are OK, which is obvious from the fact that they are empty.

Lemma 48 *The initial configuration is OK.*

```
Lemma initBDDconfig_OK : (BDDconfig_OK initBDDconfig).
```

Lemma 49 *The initial memoization table for negation is OK with respect to any configuration.*

```
Lemma initBDDneg_memo_OK_2 : (cfg:BDDconfig)
  (BDDneg_memo_OK_2 cfg initBDDneg_memo).
```

Lemma 50 *The initial memoization table for disjunction is OK with respect to any configuration.*

```
Lemma initBDDor_memo_OK : (cfg:BDDconfig)
  (BDDor_memo_OK cfg initBDDor_memo).
```

We also require that the leaf node `BDDzero` is OK in any configuration, which is obvious from the definition of when a node is OK. The same holds for the other leaf node `BDDzero`.

In a similar way we could check equivalence of two expressions by constructing up both the BDDs in the same configuration by two consecutive applications of `BDDof_bool_expr` and then checking whether two nodes are equal (using `ad_eq`). To check whether an expression is unsatisfiable, we check whether the corresponding BDD node is the leaf node `BDDzero`.

5 Experimental Results

We have conducted a series of practical experiments so as to assess:

1. Whether the Coq implementation of BDDs actually works. Although it was proved correct, it might still require so much space or time that it would not be usable for any practical problem.
2. How it scales up with problem complexity.
3. How much speed and space we gain when we extract the BDD implementation to OCaml and we run the resulting, compiled program.
4. How this compares to a C implementation of the same BDD algorithms.
5. How this compares to a C implementation of better BDD-like algorithms.

The first reassuring point is that (point 1) it indeed works. It is slow and uses lots of memory, but the point is that it actually solves practical problems.

We now examine each of the points 2–5 in turn in the following subsections. We have chosen to test our implementation on several propositional formulae, the vast majority of which being valid:

- Urquhart's U-formulae [Urq87, Pel86]: U_n is defined as $x_1 \Leftrightarrow (x_2 \Leftrightarrow \ldots \Leftrightarrow (x_n \Leftrightarrow (x_1 \Leftrightarrow (x_2 \Leftrightarrow \ldots \Leftrightarrow (x_{n-1} \Leftrightarrow x_n)\ldots)))\ldots)$. U_n is valid, and has only exponential-sized proofs using resolution.
- Pigeonhole formulae [Pel86]: pig_n states that you cannot put $n+1$ pigeons in n holes with no more than one pigeon per hole. Although this is an obvious fact of life, the formula $\forall n : \mathbb{N} \cdot pig_n$ cannot be proved in first-order logic without induction. Nonetheless, it can be expressed as a propositional formula of size $O(n^3)$.
- The 1990 IMEC benchmarks [VC90]: these are actual hardware verification benchmarks, of the form $F_1 \Leftrightarrow F_2$ or $(F_1 \Leftrightarrow F_2) \vee dcs$, where F_1 is the specification of a machine, usually described by a naive implementation, and F_2 is an implementation, usually more refined, and dcs is a possible *don't care set*. It is asked whether F_1 and F_2 indeed define the same machine, possibly modulo some states in dcs we do not care about. These tests were also used by Harrison [Har95].

While the latter problems are more realistic than the first two, the first two actually provide scalable examples, which moreover put quite some stress of the implementation. Whereas Urquhart's formulae can be solved in polynomial time by BDD implementations, pigeonhole formulae require exponential time and space.

We tested all examples on Pentium machine at 166MHz, with 96M main memory, 80M swap space, running slackware Linux 2.0.30, with libc 5.4.33. This hardware was state-of-the-art in personal computers and workstations in 1996. Our version of Coq was V6.3 (July 1999) run using `coq -full` (i.e., the native code version, as opposed to the slower bytecode version). We ran the extracted programs under Objective Caml version 2.02, using the `ocamlopt` native code compiler. To compile in C, we used gcc 2.7.2.3. We mesure times in seconds, and space in thousands of kilobytes (not to be confused with megabytes or with millions of bytes, although the difference is not too large).

5.1 Coq Speed and Space Requirements

We tested our Coq implementation of BDDs by defining various formulas F in Coq, and by calling `Time Eval Compute in (test F)`, where `test` is a function defined with the help of `BDD_of_bool_expr`, which returns the number of nodes allocated and the address of the computed BDD (which should be 1 if F is valid). The `Time` prefix is used to measure the execution time; only user times are reported. The `Eval Compute` command asks for a complete reduction of the term following the `in` keyword, following a call-by-value strategy.

Figure 1 shows the time and space requirements for our BDD implementation running under Coq, on Urquhart's formulae U_n. The formula U_n was defined in Coq by a fixpoint declaration:

```
Definition v := [n:nat](Var (ad_of_nat n)).

Fixpoint  A_bound [n,bound:nat] : bool_expr :=
  Cases bound of O => (v n)
              | (S bound') => (Iff (v (minus n bound)) (A_bound n bound'))
  end.

Definition A := [n:nat](A_bound n n).

Fixpoint U_bound [n,bound:nat] : bool_expr :=
  Cases bound of O => (A n)
              | (S bound') => (Iff (v (minus n bound')) (U_bound n bound'))
  end.

Definition U := [n:nat](U_bound n (S n)).
```

U_n is computed by invoking `(U` n`)`. This builds a formula of type `bool_expr`. Variables `(v` n`)` are ordered by their index n. In Figure 1, we have also indicated the number of BDD nodes allocated during the construction of the final BDD. The curves below show the same data: time solid, space dashed, and number of nodes dotted. After loading, and before any computation, Coq already uses 19.5 thousand kilobytes.

Regression tests show that time is about $0.113 \times n^{2.25}$, space is about $19.5 + 0.014 \times n^{1.97}$ thousand kilobytes, while the number of nodes allocated is $6.33 \times n^{1.95}$. Note that time is not quite proportional to the number of allocated nodes: this can be attributed to the fact that lookup in the state, in sharing maps and in memoizing maps is not a linear function of the state size s, rather of the form $O(n \log n)$. In any case, the behavior in time and number of nodes is roughly quadratic in n.

Pigeonhole formulae were built by the following Coq program, so that `(pb72 pigeons holes)` yields the formula that states that you cannot put `pigeons` pigeons in `holes` with at most one pigeon per hole. (The `Section` mechanism allows to declare local variables like `pigeons` and `holes`, define objects or functions like `pb72`; closing the section by `End` then builds a function mapping these local variables to the values of the objects or functions defined in the section.)

```
Definition v := [n:nat](Var (ad_of_nat n)).

Section Pigeons.
  Variable pigeons, holes : nat. (* Given numbers of pigeons and of holes, *)

  Definition holesp1 := (plus holes (1)). (* Precompute holes+1. *)

  Definition xi := [hole, pigeon:nat] (v (minus (plus (mult holes pigeon) hole)
                                                holesp1)).
  (* (xi hole pigeon) means "pigeon #pigeon is in hole #hole". *)

  Section SomeHole.
    Variable pigeon : nat. (* Now, given a pigeon,
```

n	10	20	30	40	50	60	70	80
Time	22	99	217	374	485	1159	1793	2325
Space	23.7	26.0	30.6	41.8	51.9	63.3	74.5	100.4
#nodes	570	2140	4710	8280	12850	18420	24990	32560

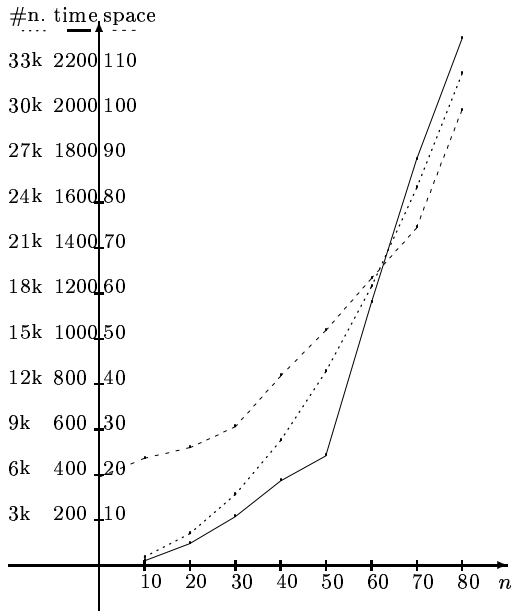


Figure 1: Urquhart's formulae U_n solved in Coq

```
    is_in_some_hole means that this pigeon is in some hole,
    i.e. (xi (1) pigeon) \/ (xi (2) pigeon) \/ ... \/ (xi holes pigeon). *)

  Fixpoint is_in_some_hole_1 [maxhole : nat] : bool_expr :=
    Cases maxhole of
        O => Zero
      | (S O) => (xi (1) pigeon)
      | (S p) => (Or (xi maxhole pigeon) (is_in_some_hole_1 p))
    end.
  Definition is_in_some_hole := (is_in_some_hole_1 holes).
End SomeHole.
(* Here, (is_in_some_hole_pigeon pigeon) denotes the fact that
   pigeon #pigeon is in some hole. *)

(* Express that each pigeon is in some hole, by
   (is_in_some_hole (1)) /\ ... /\ (is_in_some_hole pigeons). *)
Fixpoint each_pigeon_1 [pigeon : nat] : bool_expr :=
  Cases pigeon of
      O => One
    | (S O) => (is_in_some_hole (1))
    | (S p) => (And (is_in_some_hole pigeon) (each_pigeon_1 p))
  end.
Definition each_pigeon_is_in_some_hole := (each_pigeon_1 pigeons).

Section NoMore.
  Variable hole : nat. (* Given a hole number, *)

  Section G.
    Variable n : nat. (* and a pigeon #n, *)

    (* (g p) means n is not in the same hole as any pigeon numbered p or
       less:
       (~(xi hole n) \/ ~(xi hole (1))) /\ ... /\
       (~(xi hole n) \/ ~(xi hole p)).
     *)
    Fixpoint g [pigeon : nat] : bool_expr :=
      Cases pigeon of
          O => One
        | (S O) => (Or (Neg (xi hole n)) (Neg (xi hole pigeon)))
        | (S p) => (And (Or (Neg (xi hole n)) (Neg (xi hole pigeon)))
                        (g p))
      end.
  End G.
  (* In particular, (g p+1 p) means that pigeon p+1 is not in the same
     hole as any pigeon numbered p or less. *)

  (* (f maxpigeon) means that no two pigeons numbered maxpigeon at most
     occupy the same hole:
     (g (2) (1)) /\ (g (3) (2)) /\ ... /\ (g maxpigeon maxpigeon-1). *)
  Fixpoint f [maxpigeon : nat] : bool_expr :=
    Cases maxpigeon of
        O => One
      | (S O) => One
```

```
      | (S (S O)) => (g (2) (1))
      | (S p) => (And (g maxpigeon p) (f p))
    end.
  Definition has_no_more_than_one_pigeon := (f pigeons).
End NoMore.

(* Now no_hole_has_more_than_one_pigeon states what it says:
   (has_no_more_than_one_pigeon (1)) /\ ... /\
   (has_no_more_than_one_pigeon holes). *)
Fixpoint no_hole_1 [maxhole : nat] : bool_expr :=
  Cases maxhole of
      O => One
    | (S O) => (has_no_more_than_one_pigeon (1))
    | (S p) => (And (has_no_more_than_one_pigeon maxhole) (no_hole_1 p))
  end.
Definition no_hole_has_more_than_one_pigeon := (no_hole_1 holes).

Definition pb72 := (Neg (And each_pigeon_is_in_some_hole
                             no_hole_has_more_than_one_pigeon)).
```

n	1	2	3	4	5	6	7	8
Time	0.21	2.11	14.2	53.0	171.7	517.0	1617	–
Space	19.8	20.0	20.3	24.4	31.0	46.6	83.5	∞
#nodes	8	70	270	854	2498	7006	19030	–

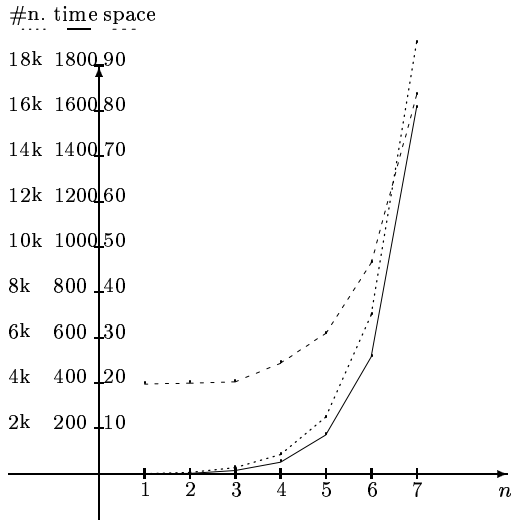


Figure 2: Pigeonhole formulae (`pb72` $n+1$ n) solved in Coq

The variable (`xi hole pigeon`) is used to represent the sentence "pigeon `pigeon` is in hole `hole`". The definition of `xi` implies the following ordering (e.g., with 3 pigeons and 2 holes): "pigeon 3 in hole 2" > "pigeon 3 in hole 1" > "pigeon 2 in hole 2" > "pigeon 2 in hole 1" > "pigeon 1 in hole 2" > "pigeon 1 in hole 1".

The results are shown in Figure 2. Seven pigeons is the limit here, and running Coq on 8 pigeons results in an out-of-memory error, denoted by the symbol ∞. Time is 0.453×3.237^n seconds, space is $19.8 + 0.141 \times 2.396^n$ thousand kilobytes, number of nodes is 15.08×2.767^n. Just after loading, Coq uses

19.8 thousand kilobytes. In any case, the behavior in time and space of the program is exponential in the number n of pigeons. Observe that the size of the n-pigeon, m-hole formula is $O(mn^2)$, so when $m = n + 1$, the pigeonhole formulae already have cubic size. According to the estimates, the 8-pigeon test would require 173 thousand kilobytes and run in 5460 seconds.

Whether we consider Urquhart's formulae or pigeonhole formulae, the BDD program running under Coq in fact exhibits a perfectly normal behavior, polynomial in the first case and exponential in the second.

We now turn to the IMEC benchmarks. They are split in four directories, `cath`, `ex`, `plasco` and `hachtel` [VC90]:

- The examples in `cath` are taken from VLSI modules realized in the Cathedral-II silicon compiler: `add1` is a 4-bit adder-subtractor, `add2` is a 4-bit ALU with 5 control signals, `add3` in an 8-bit ALU with 5 control signals, `add4` is a 12-bit ALU with 5 control signals, `addsub` compares the expressions obtained from extraction of a 14-bit logarithmic adder-subtractor and the expressions representing a 14-bit ripple adder-subtractor, and `alu` is a 4-bit ALU with 3 control signals in which 3 control-input combinations are don't cares.

  Results are shown in Figure 3. Four different BDD orderings were chosen, all simple. The two *static* orderings decide how to order variables by looking at the list of input variables, which is given in the header of each IMEC benchmark file; the *direct* static ordering numbers them 1, 2, ..., n, whereas the *inverse* static ordering numbers them n, $n-1$, ..., 1. The two *dynamic* orderings decide how to order variables by looking at the first actual occurrences of these variables in the formulae to test: in the *direct* dynamic ordering, the first variables that occurs gets number 1, the second one gets number 2, and so on; in the *inverse* dynamic ordering, they get numbers n, $n-1$, ..., 1. We abbreviate static as "S", dynamic as "D", direct as "d", inverse as "i", so we have four groups of measures labeled Sd, Si, Dd and Di. We have also added as a measure the size of the input files, counted in number of lines by the Unix utility `wc`.

Ordering	pb.	`add1`	`add2`	`add3`	`add4`	`addsub`	`alu`
Sd	Time	48.3	890.5	3125	7915	5569	198.4
	Space	23.6	44.8	85.3	89.9	38.5	30.1
	#nodes	404	5954	16438	18777	2302	1376
Si	Time	67.7	485.1	2843	10170	8262	205.7
	Space	25.6	37.4	67.7	113.5	81.0	29.9
	#nodes	862	3550	11622	22011	12110	1233
Dd	Time	63.0	410.1	–	–	5717	186.2
	Space	23.6	33.1	∞	∞	38.5	29.6
	#nodes	626	2496	–	–	2302	1051
Di	Time	62.5	431.2	–	–	8432	218.1
	Space	25.7	34.4	–	∞	80.7	30.6
	#nodes	804	2759	–	–	12122	1760
	#lines	77	164	280	388	619	222

Figure 3: Running Coq on `cath`

- The examples in `ex` are 3 to 8-bit multipliers (`mul03` through `mul08`), 2 to 8 bit ripple adders (`rip02`, `rip04`, `rip06`, `rip08`), plus various examples of diverse sizes (`ex2`, `transp`, `ztwaalf1`, `ztwaalf2`). The results are shown in Figure 4. We always used the dynamic inverse ordering on all examples.

- The examples in `plasco` deal with two-level expressions, namely expressions that are disjunctions of conjunctions of literals. Some of the files are very large: compared to, say, `ex/mul08`, which was the longest example until then and was 789 lines long in the IMEC benchmark `.be` format, `plasco/pitch` is 1181 lines long, `plasco/in1` is 2194 lines long and `plasco/cps` is 3057 lines long.

pb.	mul03	mul04	mul05	mul06	mul07	mul08	rip02	rip04	rip06	rip08
Time	38.3	352.9	2200	–	–	–	4.48	28.5	87.0	237.3
Space	22.9	31.0	56.7	∞	∞	∞	20.3	22.3	28.0	30.5
#nodes	314	2170	10135	–	–	–	80	433	1086	2039
#lines	126	213	322	453	609	789	36	54	69	86

pb.	ex2	transp	ztwaalf1	ztwaalf2
Time	1.93	1.19	36.4	27.3
Space	20.3	20.0	23.4	22.9
#nodes	37	22	482	399
#lines	27	31	38	32

Figure 4: Running Coq on ex

pb.	counter	cps	d3	hostint1	in1	mp2d	mul	pitch	rom2	table	werner
Time	21.7	–	259	15.9	–	179.0	25.0	446.9	186.9	233.6	6.73
Space	22.9	∞	35.6	21.6	∞	33.5	23.4	44.5	31.7	34.5	20.3
#nodes	247	–	2017	185	–	1852	237	2762	1607	2210	98
#lines	189	3057	611	121	2194	424	124	1181	319	325	85

Figure 5: Running Coq on plasco

The results are in Figure 5. We used the dynamic inverse ordering.

Since each file in plasco consists of several outputs that are independent of each other, i.e., since every verification in plasco is of the form $(F_1 \Rightarrow G_1) \wedge \ldots \wedge (F_n \Leftrightarrow G_n)$, where the F_is and the G_is are defined completely independently of each other, we also ran another series of tests, where we instead tested the equivalences $F_1 \Rightarrow G_1$, ..., $F_n \Leftrightarrow G_n$, separately and in sequence. Again, we used the dynamic inverse ordering on all examples. The results are shown in Figure 6. The number of nodes shown is the total number of nodes allocated. However, by separating each test $F_i \Rightarrow G_i$, we were able to start again from an empty BDD state for each new test, so that memory needs were lower.

pb.	counter	cps	d3	hostint1	in1	mp2d	mul	pitch	rom2	table	werner
Time	25.22	4368	362.8	25.1	5079	214.1	34.0	615.9	253.1	254.1	6.83
Space	23.6	74.0	33.3	23.4	72.7	31.2	23.7	37.1	30.9	32.2	20.3
#nodes	539	55823	4590	476	60602	3114	617	7782	3300	3036	138
#lines	189	3057	611	121	2194	424	124	1181	319	325	85

Figure 6: Running Coq on plasco, separating each test

Converting the big .be files of the plasco directory into Coq .v files (which was done by an automatic tool), produces huge Coq files. Consider the files generated for the tests of Figure 5. For instance, cps yields a 873.3 kilobyte file, which takes 9 minutes 14 seconds to load under the Coq toplevel. (Load times are not accounted for in the Coq running times displayed in Figures 5 and 6.) After loading, the Coq process is 60.5 thousand kilobytes large, which indicates that the sheer size of the propositional formula to examine the validity of is already of the order of 40 megabytes in Coq's main memory.

Similarly, d3 yields a 129.2 kilobyte file, takes 1 minute 45 seconds to load, and the Coq process grows to 40.0 thousand kilobytes after loading; in1 yields a 762.3 kilobyte file, takes 9 minutes 33 seconds to load and produces a 56.6 thousand kilobyte process.

- The examples in hachtel are various logic minimization examples coming from VLSI synthesis problems, and are generally easy. All tests were done using the dynamic, inverse ordering. The results are summarized in Figure 7.

pb.	alupla20	alupla21	alupla22	alupla23	alupla24	dc2	dk17	dk27
Time	114.6	174.7	422.3	456.5	224.2	57.7	70.8	29.5
Space	28.2	30.5	38.7	40.2	33.1	25.4	26.4	23.1
#nodes	1037	2098	4976	6213	3080	646	705	321
#lines	169	225	290	297	261	189	145	129

pb.	f51m	misg	mlp4	rd73	risc	root	sqn	vg2
Time	71.4	56.5	194.9	53.2	29.7	80.5	49.7	105.4
Space	27.7	26.4	31.6	25.2	23.6	27.7	24.1	28.2
#nodes	918	699	2062	621	282	793	634	1224
#lines	177	167	449	253	226	204	142	137

pb.	x1dn	x6dn	z4	z5xpl	z9sym
Time	122.7	374.3	23.1	55.7	144.5
Space	29.0	38.1	21.6	26.2	31.3
#nodes	1578	3532	310	769	1913
#lines	133	441	73	205	203

Figure 7: Running Coq on hachtel

Overall, our BDD implementation running under Coq is slow and consumes lots of memory. In fact, the main resource not to waste with BDDs is memory: the few tests that failed did so because the implementation ran out of memory. A good indicator is that our implementation, under Coq, uses between 2 and 5 kilobytes per node (around 2.5 kilobytes per node on Urquhart's formulae, around 3 to 5 kilobytes per node on pigeonhole formulae). This counts the space that the nodes themselves take, but also the space needed to represent the BDD state, the sharing maps and the memoizing tables.

Nonetheless, BDDs in Coq work on examples of quite respectable sizes, including both hard problems like Urquhart's problems or pigeonhole problems and real-life problems like the IMEC benchmarks.

Before we examine the efficiency of the extracted OCaml program, let us say that we actually cheated somehow. Indeed, a typical use of Coq to prove a propositional formula F, coded as a bool_expr, would be to state a lemma of the form Lemma my_lemma : (tauto F), where:

```
Definition tauto := [be:bool_expr]
          (Fst (Snd (BDDof_bool_expr initBDDconfig initBDDneg_memo
                                     initBDDor_memo be)))=BDDone.
```

tests whether its input boolean expression be is a tautology using BDDs. Coq will then prompt us with the goal to prove, (tauto F). Now we may either type directly Reflexivity, which will force Coq to reduce (tauto F) and check that the normal form of the left-hand side of the equality defining tauto is indeed BDDone; or we can reduce it first by typing Compute: if F is valid, Coq will reduce (tauto F) to (ad_x xH)=(ad_x xH), and Reflexivity will succeed right away. The two approaches are not equivalent, because, in the first case, (tauto F) is reduced using a default strategy that is not call-by-value. (Let us call this strategy "default".)

But in any case, Coq will tell us that `my_lemma` is proved: we then type `Qed` to save the proof, and Coq rechecks the proof. It will therefore redo all computations, using yet another reduction strategy: call the *default Qed* strategy the one used by `Qed` after we used the default strategy when looking for the proof, call the *Qed* strategy the one used by `Qed` if we used `Compute` before `Reflexivity` during proof search. The default, default Qed and Qed strategies are less efficient than call-by-value on computational tasks like our BDD implementation. We show their impact on Urquhart's and pigeonhole formulae in Figure 8 and Figure 9 respectively. We have also mentioned the time and space needed to perform a `Compute`, which turn out to be worse than the ones needed to perform an `Eval Compute`.

Slowdowns are shown as $(\times n)$ next to the times, just like increase in space usage, next to the space figures. Increase in space usage is computed as $(n - n_0)/(m - n_0)$, where m is the size of the Coq process after completion of call-by-value reduction, n is the size of the Coq process after default or Qed reduction, and n_0 is the size of the Coq process after loading and just before reduction. As can be expected, `Qed` works in time and space independent of the way proof search had been conducted, i.e., the "default Qed" and "Qed" rows are essentially the same. `Qed` works roughly 2 to 3 times slower than the `Eval Compute` evaluator and in 10 to 20 times more space. However, it is curious that the `Compute` evaluator takes so much more time and space than `Eval Compute`, knowing that they are meant to implement the same reduction strategy. The explanation (Christine Paulin-Mohring, personal communication) is that in Coq V6.3, `Compute` actually does a `Eval Compute`, then rechecks its result by applying the strategy used by `Qed`. This will be changed in later versions of Coq.

n	5	10	15	20
Default Time	28.9 (×4.6)	123.1 (×5.6)	277.2 (×5.3)	500 (×5.1)
Default Space	–	39.75 (×4.8)	56.1 (×8.3)	82.5 (×9.7)
Default Qed Time	14.8 (×3.2)	59.4 (×2.7)	137.8 (×2.7)	230.1 (×2.3)
Default Qed Space	–	40.3 (×5)	64.1 (×10.1)	95.1 (×11.6)
Compute Time	20.8 (×3.3)	94.0 (×4.3)	216 (×4.2)	382.6 (×3.9)
Compute Space	23.6	32.5 (×3.1)	50.7 (×7.1)	78.9 (×3.0)
Qed Time	14.6 (×2.3)	60.5 (×2.8)	147.1 (×2.8)	239.0 (×2.4)
Qed Space	26.7	42.0 (×5.4)	61.7 (×9.6)	87.1 (×10.4)

Figure 8: Default and Qed strategies used on Urquhart's formulae

n	1	2	3	4	5
Default Time	0.9 (×4.3)	13.8 (×6.5)	66.8 (×4.7)	252.5 (×4.8)	–
Default Space	20.0	22.6 (×14)	33.1 (×26.6)	54.7 (×7.6)	≥ 111
Default Qed Time	0.42 (×2)	6.8 (×3.2)	33.7 (×2.4)	125.0 (×2.4)	–
Default Qed Space	20.0	24.9 (×25.5)	33.1 (×26.6)	55.7 (×7.8)	–
Compute Time	0.57 (×2.7)	10.1 (×4.8)	48.7 (×3.4)	182.5 (×3.4)	579.2 (×3.4)
Compute Space	19.8	20.8 (×5)	29.0 (×18.4)	47.1 (×5.9)	109.1 (×7.8)
Qed Time	0.38 (×1.8)	7.0 (×3.3)	33.2 (×2.3)	123.7 (×2.3)	–
Qed Space	20.0	23.1 (×16.5)	30.5 (×21.4)	52.0 (×7)	≥ 112

Figure 9: Default and Qed strategies used on pigeonhole formulae

5.2 Measuring the Extracted Program

One obvious source of inefficiency in the Coq implementation of BDDs is that BDDs are constructed by running the BDD algorithms in an interpreter for a complex λ-calculus with dependent types.

We can get rid of this source of efficiency by extracting the Coq implementation, alongside the definitions of the actual formulae to prove, and get OCaml programs that we then compile and run.

We ran the same benchmark examples as in Section 5.2, this time using compiled extracted OCaml programs. The results are shown in Figures 10 through 16. When applicable, we show the speedup over the Coq implementation (row Speedup), i.e., a speedup of 114 means that the extracted program runs 114 times faster than the interpreted Coq program. Similarly, the "Sp. savings" row shows the space savings: a 10.5 figure means that the extracted program consumes 10.5 less space than the corresponding Coq program. This last figure must be taken with a grain of salt. What we do to measure the space savings is the following. First compute a, the size of the Coq process after the BDD has been constructed minus the size of the Coq process beforehand, both being measured by `ps al`. Second, measure the size b of the OCaml data by invoking `(stat ()).live_words` and multiplying by the word size (here, 4): this counts only the size of the heap that is live, i.e., in use. The space savings is counted as a/b.

n	10	20	30	40	50	60	70	80
Time	0.11	0.87	1.91	3.51	5.55	9.63	13.11	17.59
Speedup	200	114	114	107	87.4	120	137	132
Space	0.128	0.508	1.06	1.85	3.09	4.21	5.9	8.09
Sp. savings	33	12.8	10.5	12.1	10.5	15.1	12.6	12.4
#nodes	570	2140	4710	8280	12850	18420	24990	32560

n	90	100	120	150	200	250
Time	23.29	29.58	45.03	74.36	143.1	236.2
Space	10.3	12.8	18.1	26.6	51.3	71.6
#nodes	41130	50700	72840	113550	201400	314250

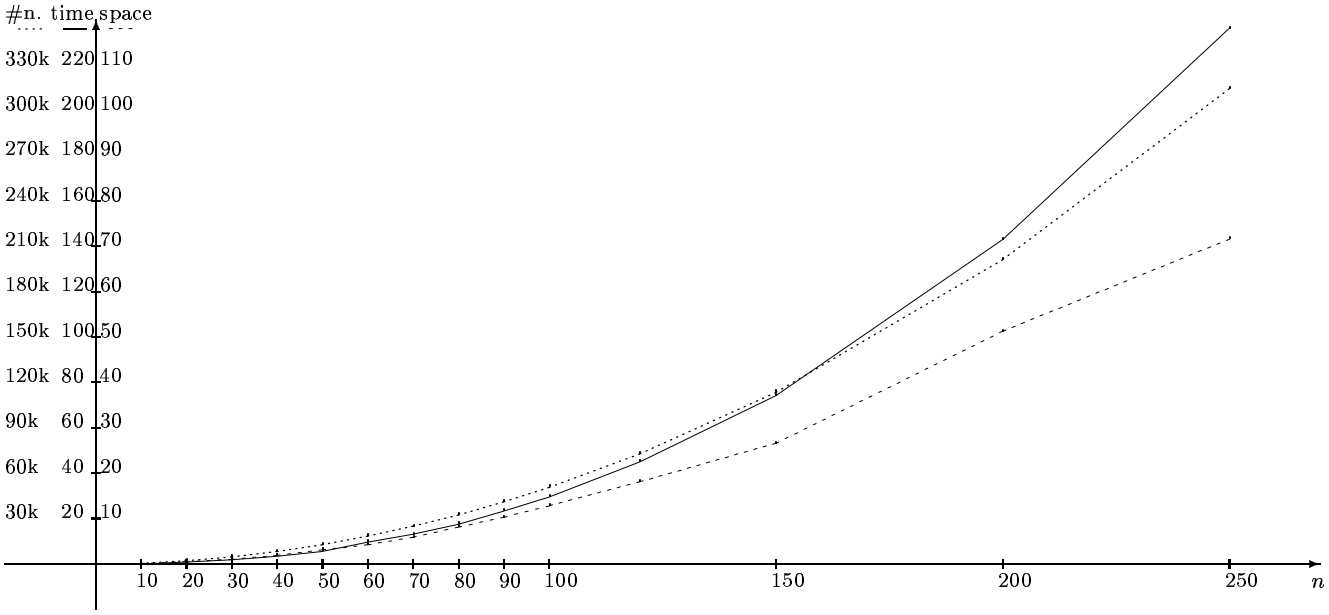


Figure 10: Urquhart's formulae U_n solved by the extracted program

Time for solving U_n is now roughly $0.000758 \times n^{2.294}$ seconds, space usage is $0.00127 \times n^{1.992}$ thousand kilobytes, while the number of nodes is about $5.945 \times n^{1.966}$. We get again a polynomial time and space behavior, with exponents close to 2. Speedup is about 130, space savings represent about a 12-fold improvement over the Coq implementation.

The results for pigeonhole formulae are shown in Figure 11. Here, some of the times are non-measurable, i.e. they fall below the value of the clock time (10 ms.). Speedup again lies around 130, and although the space savings fluctuate much more than for Urquhart's problems, they certainly are of the order of 20, ignoring small values of n for which the experimental error is too large anyway.

n	1	2	3	4	5	6	7	8	9	10
Time	0	0.02	0.08	0.27	1.15	3.75	12.0	36.8	103.9	286.1
Speedup	–	106	178	196	149	138	135	–	–	–
Space	36.10^{-6}	0.016	0.078	0.198	0.651	1.76	4.97	12.5	33.1	92.0
Sp. savings	–	12.5	6.4	23.2	17.2	26.5	16.8	–	–	–
#nodes	8	70	270	854	2498	7006	19030	50230	129162	324446

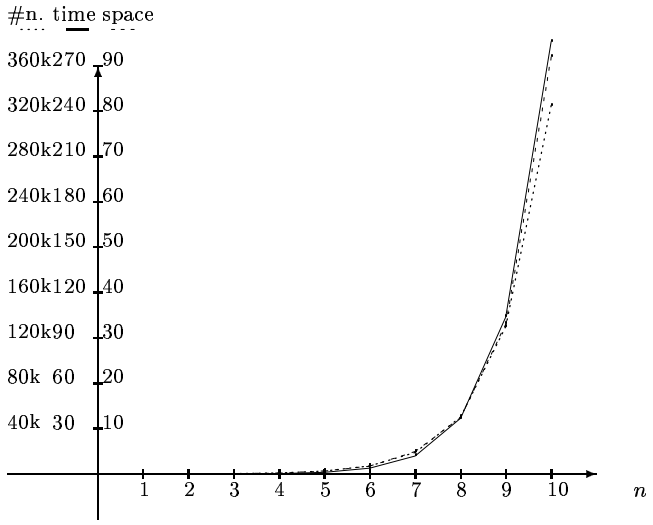


Figure 11: Pigeonhole formulae (`pb72` $n+1$ n) solved by the extracted program

The results for the `cath` examples are shown in Figure 12. We have decided to show the speedup factor next to the time, preceded by the sign÷. Speedups are now more of the order of 200. Space savings are more difficult to measure, because we have not counted exactly how much space the Coq process was using after loading each example. However, they are again of the order of 20. Note that the choice of ordering is very important in these examples. Note also that, while the Coq implementation maxes out around 30 thousand nodes, the Caml implementation is able to climb to at least 10 times more.

The tests on the `ex` examples are given in Figure 13. We only used the dynamic inverse ordering, and did not measure space usage. The only test that the OCaml implementation did not pass was the 8-bit multiplier `mul08`, which ran out of memory.

The test on `plasco` are shown in Figure 14. Notice that some of the files are so big that the extractor had trouble with them. For instance, `in1` takes 9 minutes 33 seconds to load, making Coq's process size grow to 55.3 thousand kilobytes, then takes 44 minutes and 4 seconds more to extract the OCaml file. Most of the time seems spent on traversing terms or otherwise collecting information, as the OCaml file is written during the very few seconds. Meanwhile, the size of the Coq process grows to 90 thousand kilobytes. (Moreover, we had to apply a patch, due to Jean-Christophe Filliâtre, to avoid a bug in Coq V6.3: the standard V6.3 used to truncate lines that were too long in the output OCaml code. This is corrected in

Ord.	pb.	add1	add2	add3	add4	addsub	alu
Sd	Time	0.19 (÷254)	4.64 (÷192)	18.36 (÷170)	41.67 (÷190)	23.05 (÷242)	0.89 (÷223)
	Space	0.128	1.98	4.64	5.41	0.77	0.41
	#nodes	404	5954	16438	18777	2302	1376
Si	Time	0.32 (÷212)	2.69 (÷180)	13.68 (÷208)	50.7 (÷201)	34.7 (÷238)	1.01 (÷204)
	Space	0.24	1.07	3.46	6.54	3.72	0.37
	#nodes	862	3550	11622	22011	12110	1233
Dd	Time	0.26 (÷242)	1.88 (÷218)	89.1	325.1	20.9 (÷274)	0.85 (÷219)
	Space	0.19	0.76	31.6	90.2	0.88	0.30
	#nodes	626	2496	106454	247281	2302	1051
Di	Time	0.34 (÷184)	2.42 (÷178)	39.2	–	32.1 (÷262)	1.1 (÷198)
	Space	0.22	0.87	13.2	∞	36.9	0.53
	#nodes	804	2759	40818	–	12122	1760
	#lines	77	164	280	388	619	222

Figure 12: Extracted program running on cath

pb.	mul03	mul04	mul05	mul06	mul07	mul08
Time	0.16 (÷239)	1.69 (÷209)	11.93	71.59	432.6	∞
#nodes	314	2170	10135	39252	135280	–
#lines	126	213	322	453	609	789

pb.	rip02	rip04	rip06	rip08
Time	0.01 (÷448)	0.13 (÷219)	0.4 (÷218)	1.0 (÷237)
#nodes	80	433	1086	2039
#lines	36	54	69	86

pb.	ex2	transp	ztwaalf1	ztwaalf2
Time	0.02 (÷97)	0.01 (÷119)	0.16 (÷228)	0.13 (÷210)
#nodes	37	22	482	399
#lines	27	31	38	32

Figure 13: Extracted program running on ex

later versions.) The resulting OCaml file then takes 3.46×10^6 bytes, and compiles (under `ocamlopt`) only provided the compiler's stack limit is raised above the standard value of 256 kilobytes, in 84 seconds and using 44.7 thousand kilobytes. Similarly, `cps` takes 11 minutes and 3 seconds to load, making the size of the Coq process grow to 58.6 thousand kilobytes, then 24 minutes and 32 seconds more (cpu time) to extract the OCaml program, making the size of the Coq process grow to 127 thousand kilobytes and producing an OCaml file of size 3.23×10^6 bytes. The latter file takes 34 seconds to compile under `ocamlopt`, and the `ocamlopt` process grows to 50.3 thousand kilobytes. Comparatively, executing the compiled OCaml file only takes 18.72 seconds and runs in 7.56 thousand kilobytes only. It is therefore hard to talk of a speedup in these cases; note that the times we report are pure execution times, and do not take Coq loading, extraction and compilation times into account.

pb.	`counter`	`cps`	`d3`	`hostint1`	`in1`	`mp2d`
Time	0.1 (÷217)	18.72	1.55 (÷167)	0.08 (÷199)	20.86	1.1 (÷163)
#nodes	247	20324	2017	185	24854	1852
#lines	189	3057	611	121	2194	424

pb.	`mul`	`pitch`	`rom2`	`table`	`werner`
Time	0.1 (÷250)	2.31 (÷193)	1.1 (÷170)	1.4 (÷167)	0.03 (÷224)
#nodes	237	2762	1607	2210	98
#lines	124	1181	319	325	85

Figure 14: Running the extracted program on `plasco`

As with Coq, we also did all the tests by testing each equivalence separately. The results are shown in Figure 15.

pb.	`counter`	`cps`	`d3`	`hostint1`	`in1`	`mp2d`
Time	0.11 (÷229)	23.23 (÷188)	1.87 (÷194)	0.1 (÷251)	31.77 (÷160)	1.0 (÷214)
#nodes	539	55823	4590	476	60602	3114
#lines	189	3057	611	121	2194	424

pb.	`mul`	`pitch`	`rom2`	`table`	`werner`
Time	0.16 (÷213)	2.69 (÷229)	1.23 (÷206)	1.2 (÷212)	0.03 (÷228)
#nodes	617	7782	3300	3036	138
#lines	124	1181	319	325	85

Figure 15: Running the extracted program on `plasco`, separating each test

The results on the `hachtel` examples are shown in Figure 16. These examples do not carry much new information, and do not deserve much attention, actually.

5.3 Comparing with a Hand-Written Implementation of the Same Algorithms

There are several factors that make the Coq implementation slow and consumes lots of memory. One of these, as we have just seen, is the fact that it is running in an interpreter for a complex λ-calculus.

Another is the fact that our BDD implementation actually *simulates* a store instead of using the actual memory of the computer. Even in the extracted program, the store is represented as a map, i.e., a binary

pb.	alupla20	alupla21	alupla22	alupla23	alupla24	dc2	dk17	dk27
Time	0.47 (÷244)	0.93 (÷188)	2.49 (÷170)	2.89 (÷158)	1.49 (÷150)	0.26 (÷222)	0.32 (÷221)	0.14 (÷211)
#nodes	1037	2098	4976	6213	3080	646	705	321
#lines	169	225	290	297	261	189	145	129

pb.	f51m	misg	mlp4	rd73	risc	root	sqn	vg2
Time	0.38 (÷188)	0.27 (÷209)	1.05 (÷186)	0.23 (÷231)	0.14 (÷212)	0.33 (÷244)	0.25 (÷199)	0.5 (÷211)
#nodes	918	699	2062	621	282	793	634	1224
#lines	177	167	449	253	226	204	142	137

pb.	x1dn	x6dn	z4	z5xpl	z9sym
Time	0.7 (÷175)	2.05 (÷183)	0.12 (÷193)	0.33 (÷169)	0.85 (÷170)
#nodes	1578	3532	310	769	1913
#lines	133	441	73	205	203

Figure 16: Running the extracted program on `hachtel`

tree, and reading at a given address is simulated by descending through this binary tree in search of the entry that represents this address. To test the impact of this simulation, we coded the same BDD algorithms in C, using the standard C library's memory allocation primitive `malloc` (except we use `malloc` to allocate chunks of 1024 nodes at a time, and allocate from these chunks), and replacing all simulated reads by actual memory reads. Sharing maps and memoization maps are implemented as simple hash tables, with 23227 entries, each entry being a list of nodes with the same hash code; the hash code of a node is the sum of its variable and the addresses of its left and right sons, modulo 23227.

The results are shown in Figures 17 through 22. Speedups and space savings are measured relative to the extracted OCaml program of Section 5.2. Note that speedups degrade as the number of nodes increase: this is due to the fact that our hash tables in C do not expand as the number of nodes grow, where sharing and memoization maps do. In Figures 19 to 22, we have systematically used the dynamic inverse ordering. Times shown as 0 denote non-measurable times, typically under 10ms. Note that, this time, `mul08` was solved, in 8 minutes 23 seconds, using 17.7 thousand kilobytes. Applying the rule of thumb, witnessed by the other tests, that the OCaml implementation needs about 6 times as much space, this places the verification of the 8-bit multiplier in the reach of machines with at least 116.2 thousand kilobytes. Also, the times measured on the C program still only apply to conversion from a boolean expression to a BDD, not to parsing or the construction of the boolean expression. However, the times of the latter actions are usually of the order of several hundred milliseconds at most. This makes the times for `in1` and `cps` in Figure 21 particularly dramatic, as parsing, extracting to OCaml and compiling took times of the order of an hour, wheras here parsing and building the boolean expression whose BDD we are interested in basically take no time.

In general, speedups vary wildly compared to the OCaml implementation, from about 10 to 280. On the other hand, space savings lie around 6.

5.4 Changing the Algorithms

There are still a few sources of inefficiency in the C program of Section 5.3. One is the fact that, if it encounters twice the same boolean expression, it will compute twice the same BDD: boolean expressions are not memoized. This is traditionally solved partially by not building any boolean expression, and computing the BDD directly while parsing. This indeed solves the case where the benchmark files define some boolean variable x as some expression $f(y)$, where y occurs twice, and y is defined as some formula F. Until now, we actually replaced y by F textually, yielding a definition of x as $f(F)$: this makes the program compute the BDD of F twice. It is more reasonable to compute the BDD of F, store it in y, then reuse this BDD in the computation of the BDD of $f(y)$ each time we need that of y.

n	10	20	30	40	50	60	70	80
Time	0	0	0.02	0.05	0.02	0.07	0.11	0.12
Speedup	∞	∞	95	70	278	138	119	147
Space	0.057	0.119	0.180	0.303	0.442	0.623	0.848	1.085
Sp. savings	2	4.3	5.9	6.1	7.0	6.8	7.0	7.5

n	100	120	150	200	250
Time	0.26	0.43	0.64	1.42	2.68
Speedup	114	105	116	101	88
Space	1.69	2.42	3.74	6.63	10.33
Sp. savings	7.6	7.5	7.1	7.7	6.9

n	300	400	500	600	700	800
Time	4.32	11.35	26.28	60.38	116.85	206.64
Space	14.86	26.36	41.15	59.22	80.58	105.2

Figure 17: Urquhart's formulae U_n solved in C

n	1	2	3	4	5	6	7	8	9	10	11	12
Time	0	0	0	0.01	0.03	0.06	0.12	0.57	1.78	6.92	36.01	247.8
Speedup	–	–	–	27	38	63	100	65	58	41	–	–
Space	0.057	0.057	0.057	0.057	0.119	0.315	0.795	2.05	5.23	13.09	32.14	77.66
Sp. savings	–	0.28	1.37	3.47	5.47	5.59	6.25	6.1	6.3	7.0	–	–

Figure 18: Pigeonhole formulae (`pb72` $n+1$ n) solved in C

pb.	`add1`	`add2`	`add3`	`add4`	`addsub`	`alu`
Time	0.01 (÷26)	0.05 (÷38)	0.51 (÷175)	22.43 (÷14.5)	1.15 (÷18.2)	0.02 (÷42)
Space	0.057 (÷3.3)	0.156 (÷4.9)	1.68 (÷18.8)	23.1 (÷3.9)	0.467 (÷1.9)	0.094 (÷3.2)
#lines	77	164	280	388	619	222

Figure 19: C program running on `cath`

pb.	mul03	mul04	mul05	mul06	mul07	mul08
Time	0.01 (÷16)	0.04 (÷42)	0.33 (÷36)	2.86 (÷25)	34.56 (÷12.5)	502.6
#lines	126	213	322	453	609	789

pb.	rip02	rip04	rip06	rip08
Time	0	0	0.02 (÷20)	0.01 (÷100)
#lines	36	54	69	86

pb.	ex2	transp	ztwaalf1	ztwaalf2
Time	0	0	0	0
#lines	27	31	38	32

Figure 20: C program running on ex

pb.	counter	cps	d3	hostint1	in1	mp2d
Time	0	0.21 (÷89)	0.03 (÷52)	0	0.30 (÷69.5)	0.02 (÷55)
#lines	189	3057	611	121	2194	424

pb.	mul	pitch	rom2	table	werner
Time	0	0.04 (÷58)	0.02 (÷55)	0.01 (÷140)	0
#lines	124	1181	319	325	85

Figure 21: Running the C program on plasco

pb.	alupla20	alupla21	alupla22	alupla23	alupla24	dc2	dk17	dk27
Time	0	0.03 (÷31)	0.04 (÷62)	0.02 (÷145)	0.01 (÷149)	0.01 (÷26)	0.02 (÷16)	0.01 (÷14)
#lines	169	225	290	297	261	189	145	129

pb.	f51m	misg	mlp4	rd73	risc	root	sqn	vg2
Time	0.01 (÷38)	0	0.02 (÷53)	0	0	0.01 (÷33)	0.01 (÷25)	0.01 (÷50)
#lines	177	167	449	253	226	204	142	137

pb.	x1dn	x6dn	z4	z5xpl	z9sym
Time	0.01 (÷70)	0.04 (÷51)	0.01 (÷12)	0.01 (÷33)	0.02 (÷43)
#lines	133	441	73	205	203

Figure 22: Running the C program on hachtel

A second source of inefficiency is the fact that we have used one of the most elementary BDD algorithm. One standard improvement is to use *typed decision graphs* (TDGs, [Bil87]), whose size is lower than BDDs and which allow one to compute negations in constant time.

Urquhart's problems and pigeonhole formulae show strictly no improvement using the first trick. This is normal, since they do not suffer from the duplication problem. However, some other problems do, notably the multipliers of `ex`: see Figure 24, where speedups are shown relative to the C program of Section 5.3. But some problems get slower as well, see Figure 23 and Figure 25: this can be attributed to the fact that the times shown here also include parsing times, which were not taken into account in the previous tests. Space usage is not shown, but is more or less the same as in Section 5.3, which should not be surprising, since the same BDDs are being built as in Section 5.3.

Applying the first improvement can in principle be done at the Coq level, however this would need some reengineering of the code and the proofs, which we have not done.

pb.	`add1`	`add2`	`add3`	`add4`	`addsub`	`alu`
Time	0.01 (÷1)	0.02 (÷2.5)	0.29 (÷1.76)	26.18 (÷0.86)	0.10 (÷11.5)	0.02 (÷1)
Space	0.094 (÷0.6)	0.172 (÷0.9)	1.53 (÷1.1)	27.9 (÷0.83)	0.467 (÷1)	0.135 (÷0.7)
#lines	77	164	280	388	619	222

Figure 23: C program running on `cath`, first improvement

pb.	`mul03`	`mul04`	`mul05`	`mul06`	`mul07`	`mul08`
Time	0	0.02 (÷2)	0.08 (÷4.1)	0.31 (÷9.2)	1.75 (÷19.7)	13.64 (÷36.8)
#lines	126	213	322	453	609	789

pb.	`rip02`	`rip04`	`rip06`	`rip08`
Time	0	0.01	0.01 (÷2)	0.01 (÷1)
#lines	36	54	69	86

Figure 24: C program running on `ex`, first improvement

pb.	`counter`	`cps`	`d3`	`hostint1`	`in1`	`mp2d`
Time	0.01	0.49 (÷0.43)	0.09 (÷0.33)	0.02	0.56 (÷0.54)	0.06 (÷0.33)
#lines	189	3057	611	121	2194	424

pb.	`mul`	`pitch`	`rom2`	`table`	`werner`
Time	0	0.10 (÷0.4)	0.06 (÷0.33)	0.07 (÷0.14)	0
#lines	124	1181	319	325	85

Figure 25: Running the C program on `plasco`, first improvement

The influence of the second improvement, namely switching to TDGs, is shown in the following figures, where speedups are given relative to the C program of Section 5.3. (The first improvement was also applied.)

n	30	40	50	60	70	80
Time	0.01	0.05	0.06	0.05	0.05	0.07
Speedup	2	1	0.3	1.4	2.2	1.7
Space	0.101	0.183	0.264	0.346	0.469	0.592
Sp. savings	1.8	1.66	1.67	1.80	1.81	1.83

n	100	120	150	200	250
Time	0.13	0.24	0.39	0.84	1.54
Speedup	2.0	1.8	1.64	1.7	1.74
Space	0.920	1.31	2.03	3.56	5.55
Sp. savings	1.84	1.85	1.84	1.86	1.86

n	300	400	500	600	700	800	900	1000
Time	2.70	6.73	15.43	30.23	61.12	114.17	179.56	277.47
Speedup	1.6	1.7	1.7	2.0	1.9	1.8	–	–
Space	7.99	14.15	22.1	31.78	43.23	56.46	71.43	88.16
Sp. savings	1.86	1.86	1.86	1.86	1.86	1.86	–	–

Figure 26: Urquhart's formulae U_n solved in C, with TDGs

n	1	2	3	4	5	6	7	8	9	10	11	12	13
Time	0	0	0	0	0.03	0.01	0.08	0.20	0.59	2.56	11.93	65.61	338.3
Speedup	–	–	–	–	1	6	1.5	2.8	3.0	2.7	3.0	3.8	–
Space	0.041	0.041	0.041	0.041	0.082	0.164	0.410	1.04	2.64	6.59	16.18	39.06	92.81
Sp. savings	1.4	1.4	1.4	1.4	1.45	1.9	4.85	2.0	2.0	2.0	2.0	2.0	–

Figure 27: Pigeonhole formulae (`pb72` $n+1$ n) solved in C, with TDGs

pb.	`add1`	`add2`	`add3`	`add4`	`addsub`	`alu`
Time	0	0.02 (÷2.5)	0.18 (÷2.8)	11.94 (÷1.9)	0.05 (÷23)	0.02 (÷1)
Space	0.077 (÷0.74)	0.119 (÷1.3)	0.88 (÷1.9)	15.3 (÷1.5)	0.291 (÷1.6)	0.098 (÷0.96)
#lines	77	164	280	388	619	222

Figure 28: C program running on `cath`, using TDGs

pb.	mul03	mul04	mul05	mul06	mul07	mul08
Time	0.01 (÷1)	0.01 (÷4)	0.04 (÷8.3)	0.21 (÷13.6)	1.08 (÷32)	8.11 (÷62)
#lines	126	213	322	453	609	789

pb.	rip02	rip04	rip06	rip08
Time	0	0.01	0.02 (÷1)	0.01 (÷1)
#lines	36	54	69	86

pb.	ex2	transp	ztwaalf1	ztwaalf2
Time	0	0	0	0
#lines	27	31	38	32

Figure 29: C program running on ex, using TDGs

pb.	counter	cps	d3	hostint1	in1	mp2d
Time	0	0.45 (÷0.47)	0.10 (÷0.33)	0	0.58 (÷0.52)	0.02 (÷1)
#lines	189	3057	611	121	2194	424

pb.	mul	pitch	rom2	table	werner
Time	0	0.12 (÷0.33)	0.03 (÷0.67)	0.07 (÷0.14)	0
#lines	124	1181	319	325	85

Figure 30: Running the C program on plasco, with TDGs

TDGs therefore bring up a factor of two improvement in speed and in space, roughly. Note that in no implementation we reclaimed any memory: any allocated object remains allocated until the end of the program. This means that actual implementations may use much less nodes, hence less memory to solve the same problems. However, actual implementations that reclaim memory also require more space per node to store the data required for managing memory.

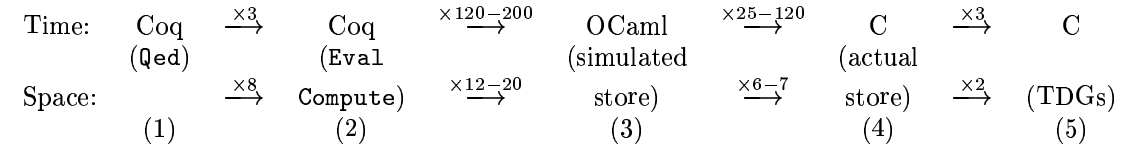
6 Conclusion

We have shown that it was possible to implement actual BDD algorithms that run in Coq, and that can be extracted to yield certified BDD implementations. We have described the formal Coq proof of the algorithms. It is long and requires the statement of many invariants, some of which are not mentioned in standard correctness proofs. Notably, the fact that the semantics of BDD nodes that existed before an operation remains the same after the operation, or that computing the disjunction preserves the validity of the memoization table for negation, are required but usually overlooked. In this respect, our work confirms the fact that BDD algorithms are correct.

Computing BDDs inside Coq is not fast, but it works and is safe. It is even able to solve real-life industrial problems, as we hope to have demonstrated. In particular, one possible use of BDDs in Coq is as follows. Imagine a complex hardware and software system that you have to validate, where you would like to model hardware components via BDDs, combinations of hardware modules in higher-order logic, and software in Coq, by abstracting hardware as abstractly specified modules. Our implementation allows you to do all this inside Coq itself. Otherwise, you would have to use an external model-checker for hardware components, and somehow trust the results of the model-checker when translating them to Coq for verifying the hardware and software modules on top of these components.

On the other hand, the value of model-checkers is that they give you explicit counterexamples when your design is flawed. It is natural to use a fast, external model-checker to this end. When your design is correct, you can then make Coq recheck that it is correct using the BDD algorithm implemented in Coq. This takes time, but you only do it once.

Talking about speed and space, we can sum up our practical experiments by the following diagram:



where the figures shown are rough improvements in time and space used. These figures can only improve, i.e., become lower, for a number of reasons:

- From (1) to (2): the speed of the Coq `Qed` evaluator is mostly important when checking proofs with large computational content, as our BDD proofs. It is therefore interesting to simply make it use the same evaluation strategy as `Eval Compute`. So `Qed` can be made as fast as `Eval Compute`. Other possibilities are to parameterize `Qed` by names of reduction strategies, or have `Qed` reduce terms according to the strategy that was used when developing the proof through tactics. In any case, future versions of Coq will undoubtedly make column (1) as fast as column (2), and use not much more memory.

- From (2) to (3): this gives an estimate that the the Coq λ-calculus interpreter is currently roughly 120 to 200 times slower than a good compiler for a mostly functional language, OCaml. Also, there is quite an overhead in the way Coq represents elements of datatypes (from 12 to 20). This is due to the fact that types are represented explicitly in Coq terms, and to the fact that terms themselves are represented using heavy but generic term representation datatypes. The problem of explicit representation of types in terms might be solved in the future by switching to a calculus of inductive constructions with implicit typing, as studied currently by Alexandre Miquel and Hugo Herbelin, both of the Coq team.

  There are two ways that Coq can be made as fast as OCaml here. The first is to have Coq trust the output of programs extracted from proofs: we could then execute the extracted OCaml program, then reimport the output of the OCaml program inside Coq. This would then make Coq as fast as OCaml,

and at the time use no more memory than OCaml. This is actually harder than it seems, first because the extraction process has to be done in such a way that we are guaranteed that the extracted program has the same behavior as the Coq program; that is, the linking facilities of Coq notions with OCaml identifiers has to be controlled severely; and second because it requires a systematic and trustable way of translating Coq input to OCaml input. The NuPRL experience shows that this is actually quite a task.

The second way to make Coq as fast as OCaml is to actually change the Coq interpreter so that it instead compiles the term to evaluate at run-time, and executes the resulting compiled code. This technique will not solve the space consumption problem, but gives some non-negligible improvements in speed. Experiments along this line have already been conducted by Henri Laulhère, a former member of the Coq team, who got speedups of about 30 on small examples. Hugo Herbelin, of the Coq team, is currently exploring this technique further.

- From (3) to (4): this shows that the penalty for simulating the store instead of using the actual store of the computer at hand is high, assuming that the difference in efficiency between OCaml and C is negligible. Better representations of maps, i.e., improving the `Map` datatype might offer some performance improvement, but it seems implausible that we will get much from such ideas. Another possibility would be to have native memoizing functions in Coq. That is, if we had `Fixpoint` declarations with a decoration stating that the defined fixpoint is memoizing (i.e., stores previously computed results in a table that it consults before executing the code), we could dispense with our sharing maps and our memoization maps for negation and disjunction. Furthermore, if the Coq run-time systematically shared all data in memory, as HimML does [Gou94], we could represent BDDs not as addresses but as actual objects of a recursive datatype, and the correctness proof would be much simpler. It is however not clear that such a radical change of implementation for Coq would be of a great benefit outside the domain of BDDs.

- From (4) to (5). This shows that using TDGs instead of ordinary BDDs brings relatively little. However, it should be feasible to implement TDGs or more complicated shared graph structures in Coq and prove this implementation correct. This would be interesting in any case.

Apart from performance improvement perspectives, there are two natural continuations of this work. First, some form of garbage-collection has to be implemented if we ever wish to use BDDs in Coq in a serious way. Reference counting is probably too hard to do, as complex invariants would have to be carried around each proof, including proofs of lemmas not concerned at all with memory management. Implementing a mark-and-sweep collector as HimML's [Gou94] seems less intrusive. Second, it should not be too hard to extend this work to build a complete symbolic model-checker in Coq on top of our BDD implementation, provided garbage-collection has been added first.

Acknowledgements

We thank John Harrison for providing us the IMEC benchmark suite in record time.

References

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 95–105, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[BBC+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.

[BC93] D. A. Basin and R. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993. Also available as Technical Report MPI-I-92-205.

[Bil87] Jean-Paul Billon. Perfect normal forms for discrete functions. Technical Report 87019, Bull S.A. Research Center, June 1987.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, New York, 1979.

[BM81] Robert Boyer and J. Strother Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, London, 1981. Academic Press.

[Bor97] Arne Borälv. The Industrial Success of Verification Tools Based on Stålmarck's Method. In Orna Grumberg, editor, *Proceedings of the Ninth International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10. Springer Verlag, 1997.

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. LNCS, Springer-Verlag, 1997.

[Bry86] Randall E. Bryant. Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, C35(8):677–692, August 1986.

[Bry99] Randall E. Bryant. Microprocessor verification using efficient decision procedures for a logic of equality with uninterpreted functions. In *International Conference on Analytic Tableaux and Related Methods.* Springer Verlag LNAI 1617, Saratoga Springs, NY, June 1999.

[Cou91] Olivier Coudert. *SIAM : Une Boîte à Outils Pour la Preuve Formelle de Systèmes Séquentiels.* PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, October 1991.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[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.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[GMW77] Michael J. C. Gordon, Robin Milner, and Chris Wadsworth. Edinburgh LCF, a mechanical logic of computation. Report CSR-11-77 (in 2 parts), Department of Computer Science, University of Edinburgh, 1977.

[Gou94] Jean Goubault. Standard ML with fast sets and maps. In *5th ACM SIGPLAN Workshop on ML and its Applications (ML'94).* ACM Press, Orlando, FL, USA, June 1994.

[Har95] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.

[Har96] John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234, Turku, Finland, 1996. Springer Verlag.

[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`.

[Knu73] D. E. Knuth. *The Art of Computer Programming III: Sorting and Searching.* Addison–Wesley, Reading, Massachusetts, 1973.

[KSK93] Ramayya Kumar, Klaus Schneider, and Thomas Kropf. Structuring and automating hardware proofs in a higher-order theorem-proving environment. *Journal of Formal System Design*, pages 165–230, 1993.

[Led93] Emmanuel Ledinot. Canonicity of binary decision dags. Part of the Coq contribs, available at `http://pauillac.inria.fr/coq/contribs/bdd.html`, March 1993.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[Moo94] J. Moore. Introduction to the OBDD algorithm for the ATP community. *Journal of Automated Reasoning*, 12:33–45, 1994.

[Oos96] Maartijn Oostdijk. Proof by calculation. Master's thesis, Technische Universiteit Eindhoven, August 1996.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE'92)*, pages 748–752. Springer Verlag LNAI 607, Saratoga Springs, June 1992.

[Pel86] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2:191–216, 1986. Errata: Pelletier, Francis Jeffry, JAR 4, pp. 235–236 and Pelletier, Francis Jeffry and Sutcliffe, Geoff, JAR 18(1), p. 135.

[Pos93] Joachim Posegga. *Deduktion mit Shannongraphen für Prädikatenlogik erster Stufe.* Infix Verlag, Sankt Augustin, 1993.

[Stå94] Gunnar Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula. United States Patent number 5,276,897, 1994.

[Urq87] Alasdair Urquhart. Hard examples of resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[US94] Tomás Uribe and Mark E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. *Lecture Notes in Computer Science*, 845, 1994.

[VC90] Diederik Verkest and Luc Claesen. Special benchmark session on tautology checking. In Luc Claesen, editor, *IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, pages 81–82, 1990.

[Wey80] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artifical Intelligence*, 13(1, 2):133–170, 1980.

[YL97] Shenwei Yu and Zhaohui Luo. Implementing a model checker for LEGO. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 442–458. Springer-Verlag, September 1997. ISBN 3-540-63533-5.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399