# Formal Verification of Bit-Vector Invertibility Conditions in Coq

Burak Ekici[1], Arjun Viswanathan[2], Yoni Zohar[3(✉)], Cesare Tinelli[2], and Clark Barrett[4]

[1] Muğla Sıtkı Koçman University, Muğla, Turkey
[2] The University of Iowa, Iowa City, USA
[3] Bar-Ilan University, Ramat Gan, Israel
yoni.zohar@biu.ac.il
[4] Stanford University, Stanford, USA

**Abstract.** We prove the correctness of invertibility conditions for the theory of fixed-width bit-vectors—used to solve quantified bit-vector formulas in the Satisfiability Modulo Theories (SMT) solver cvc5— in the Coq proof assistant. Previous work proved many of these in a completely automatic fashion for arbitrary bit-width; however, some were only proved for bit-widths up to 65, even though they are being used to solve formulas over larger bit-widths. In this paper we describe the process of proving a representative subset of these invertibility conditions in Coq. In particular, we describe the `BVList` library for bit-vectors in Coq, our extensions to it, and proofs of the invertibility conditions.

## 1 Introduction

Many applications in hardware and software verification rely on bit-precise reasoning, which can be modeled using the SMT-LIB 2 theory of fixed-width bit-vectors [3]. While Satisfiability Modulo Theories (SMT) solvers are able to reason about bit-vectors of fixed width, they currently require all widths to be expressed concretely (by a numeral) in their input formulas. For this reason, they cannot be used to prove properties of bit-vector operators that are parametric in the bit-width, such as the associativity of bit-vector concatenation. Proof assistants such as Coq [25], which have direct support for dependent types, are better suited for such tasks.

Bit-vector formulas that are parametric in the bit-width arise in the verification of parametric Boolean functions and circuits (see, e.g., [13]). In our case, we are mainly interested in parametric lemmas that are relevant to internal techniques of SMT solvers for the theory of fixed-width bit-vectors. These include, for example, rewrite rules, refinement schemes, and preprocessing passes. Such techniques are developed a priori for every possible bit-width. Meta-reasoning about the correctness of such solvers then requires bit-width independent reasoning.

In this paper, we focus on parametric lemmas that originate from a quantifier-instantiation technique implemented in the SMT solver cvc5 [2]. This technique


U. Sattler and M. Suda (Eds.): FroCoS 2023, LNAI 14279, pp. 41–59, 2023.
https://doi.org/10.1007/978-3-031-43369-6_3

is based on *invertibility conditions* [15]. For a trivial case of an invertibility condition, consider the equation $x + s = t$. where $x$, $s$ and $t$ are variables of the same bit-vector sort. In the terminology of Niemetz et al. [15], this equation is "invertible for $x$." A general inverse, or "solution," is given by the term $t - s$. Since there is always such an inverse, the invertibility condition for $x + s = t$ is simply the universally true formula $\top$. The formula stating this fact, referred to here as an *invertibility equivalence*, is $\top \Leftrightarrow \exists x.\ x + s = t$, which is valid in the theory of fixed-width bit-vectors, for any bit-width. In contrast, the equation $x \cdot s = t$ is not always invertible for $x$. A necessary and sufficient condition for invertibility in this case was found in [15] to be $(-s \mid s)\ \&\ t = t$. So, the invertibility equivalence $(-s \mid s)\ \&\ t = t \Leftrightarrow \exists x.\ x \cdot s = t$ is valid for any bit-width. Notice that the invertibility condition does not contain $x$. Hence, invertibility conditions can be seen as a technique for quantifier elimination.

In [15], a total of 160 invertibility conditions were provided. However, they were verified only for bit-widths up to 65, due to the reasoning limitations of SMT solvers mentioned earlier. Recent work [16,17] addresses this challenge by translating the invertibility equivalences to the combined theory of non-linear integer arithmetic and uninterpreted functions. This approach was partially successful, but failed to verify over a quarter of the equivalences.

We verify invertibility equivalences proposed in [15] by proving them interactively in Coq. From a representative subset of the invertibility equivalences, we prove 19 equivalences, 12 of which were not proven in [16,17]. For the remaining 7, that were already proved there, our Coq proofs provide more confidence. Our results offer evidence that proof assistants can support automated theorem provers in meta-verification tasks. To facilitate the verification of invertibility equivalences, we use a rich Coq library for bit-vectors, which is a part of the SMTCoq project [10]. This Coq library models the theory of fixed-width bit-vectors adopted by the SMT-LIB 2 standard [3]. For this work, we extended the library with the arithmetic right-shift operation and the unsigned weak less-than

**Table 1.** The signatures $\Sigma_1$ and $\Sigma_0$ with SMT-LIB 2 syntax. $\Sigma_1$ consists of the operators in the entire table. $\Sigma_0$ consists of the operators in the upper part.

| Symbol | SMT-LIB Syntax | Sort |
|---|---|---|
| $=, \neq$ | =, distinct | $\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$ |
| $<_u, >_u, \leq_u, \geq_u$ | bvult, bvugt, bvule, bvuge | $\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$ |
| $\sim, -$ | bvnot, bvneg | $\sigma_{[n]} \to \sigma_{[n]}$ |
| $\&, \mid, \ll, \gg, \gg_a$ | bvand, bvor, bvshl, bvlshr, bvashr | $\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$ |
| $+$ | bvadd | $\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$ |
| $<_s, >_s, \leq_s, \geq_s$ | bvslt, bvsgt, bvsle, bvsge | $\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$ |
| $\cdot$, mod, $\div$ | bvmul, bvurem, bvudiv | $\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$ |
| $\circ$ | concat | $\sigma_{[n]} \times \sigma_{[m]} \to \sigma_{[n+m]}$ |
| $[u : l]$ | extract | $\sigma_{[n]} \to \sigma_{[u-l+1]}$ |

and greater-than predicates. To summarize, the contributions of this paper are as follows: (*i*) a description of the SMTCoq bit-vector library; (*ii*) extensions to the signature and proofs of the library; and (*iii*) formal proofs in Coq of invertibility equivalences. These contributions, while important in their own right, have the potential to go beyond the verification of invertibility equivalences. For (*i*) and (*ii*), we envision that the library, as well as its extension, will be useful for the formalization of other bit-precise reasoning mechanisms, especially related to SMT, such as rewriting rules, lemma schemas, interactive verification, and more. For (*iii*), invertibility conditions are primarily used for quantifier instantiation (see, e.g., [15]). We hope that the increased confidence in their correctness will encourage their usage in other contexts and in more solvers. Further, the formal proofs can serve as guiding examples for other proofs related to bit-precise reasoning.

The remainder of this paper is organized as follows. After technical preliminaries in Sect. 2, we formalize invertibility conditions in Sect. 3 and discuss previous attempts at verifying them. In Sect. 4, we describe the Coq library and our extensions to it. In Sect. 5, we discuss our Coq proofs. We conclude in Sect. 6 with directions for future work. A preliminary version of this work was presented as an extended abstract in the proceedings of the PxTP 2019 workshop [11]. The current version is more detailed and complete. In particular, the one Coq proof that was missing in [11] is now completed.

## 2 Preliminaries

### 2.1 Theory of Bit-Vectors

We assume the usual terminology of many-sorted first-order logic with equality (see, e.g., [12]). We denote equality by $=$, and use $x \neq y$ as an abbreviation for $\neg(x = y)$. The signature $\Sigma_{BV}$ of the SMT-LIB 2 theory of fixed-width bit-vectors defines a unique sort for each positive integer $n$, which we denote by $\sigma_{[n]}$. For every positive integer $n$ and bit-vector of width $n$, the signature contains a constant symbol of sort $\sigma_{[n]}$, representing that bit-vector, which we denote as a binary string of length $n$. The function and predicate symbols of $\Sigma_{BV}$ are as described in the SMT-LIB 2 standard. Formulas of $\Sigma_{BV}$ are built from variables, bit-vector constants, and the function and predicate symbols of $\Sigma_{BV}$, along with the usual logical connectives and quantifiers. We write $\psi[x_1, \ldots, x_n]$ to represent a formula whose free variables are from the set $\{x_1, \ldots, x_n\}$.

The semantics of $\Sigma_{BV}$-formulas is given by interpretations where the domain of $\sigma_{[n]}$ is the set of bit-vectors of width $n$, and the function and predicate symbols are interpreted as specified by the SMT-LIB 2 standard. A $\Sigma_{BV}$-formula is *valid* in the theory of fixed-width bit-vectors if it is satisfied by every such interpretation.

Table 1 contains the operators from $\Sigma_{BV}$ for which invertibility conditions were defined in [15]. We define $\Sigma_1$ to be the signature that contains only these symbols. $\Sigma_0$ is the sub-signature obtained by only taking the operators from the

upper part of the table. We use the (overloaded) constant 0 to represent the bit-vectors composed of all 0-bits.

### 2.2 Coq

The Coq proof assistant is based on the calculus of inductive constructions (CIC) [20]. It implements properties as types, and proofs as terms, reducing proof-checking to type-checking. Coq has a rich type system, that allows for highly expressive propositions to be stated and proved in this manner. One particular feature of interest is that of *dependent types* — types that can depend on values — through which one can express correctness properties within types. We refer to non-dependent types as *simple types*.

The Coq module system — in addition to allowing for principled separations of large developments — allows the abstraction of complex types along with operations over them as *modules*. A *module signature* or *module type* acts as an interface to a module, specifying the type it encapsulates along with the signatures of the associated operators. A *functor* is a module-to-module function.

## 3 Invertibility Conditions and Their Verification

In [15], a technique to solve quantified bit-vector formulas is presented, which is based on *invertibility conditions*.

**Definition 1.** *An invertibility condition for a variable $x$ in a $\Sigma_{BV}$-literal $\ell[x,s,t]$ is a formula $IC[s,t]$ such that $\forall s.\forall t.\ IC[s,t] \Leftrightarrow \exists x.\ \ell[x,s,t]$ is valid in the theory of fixed-width bit-vectors.*

*Example 1.* The invertibility condition for $x$ in $x \,\&\, s = t$ is $t \,\&\, s = t$. □

In [15], invertibility conditions are defined for a representative set of literals $\ell$ over the bit-vector operators of $\Sigma_1$, having a single occurrence of $x$. The soundness of the technique proposed in that work relies on the correctness of the invertibility conditions. Every literal $\ell[x,s,t]$ and its corresponding invertibility condition $IC[s,t]$ induce an *invertibility equivalence*.

**Definition 2.** *The* invertibility equivalence *associated with the literal $\ell[x,s,t]$ and its invertibility condition $IC[s,t]$ is the formula*

$$IC[s,t] \Leftrightarrow \exists x.\ \ell[x,s,t] \tag{1}$$

The correctness of invertibility equivalences should be verified for all possible sorts for the variables $x, s, t$ for which the condition is well sorted. Concretely, one needs to prove the validity of the following formula:

$$\forall n : \mathbb{N}.\ n > 0 \Rightarrow\ \forall s : \sigma_{[n]}.\forall t : \sigma_{[n]}.\ IC[s,t] \Leftrightarrow \exists x : \sigma_{[n]}.\ \ell[x,s,t] \tag{2}$$

This was done in [15], but only for concrete values of $n$ from 1 to 65, using solvers for the theory of fixed-width bit-vectors. In contrast, Eq. (2) cannot even

be expressed in this theory. To overcome this limitation, later work suggested a translation from bit-vector formulas over *parametric* bit-widths to the theory of non-linear integer arithmetic with uninterpreted functions [16,17]. Thanks to this translation, the authors were able to verify the correctness of 110 out of 160 invertibility equivalences. For the remaining 50 equivalences, it then seems appropriate to use a proof-assistant, as this allows for more intervention by the user who can provide crucial intermediate steps. Even for the 110 invertibility equivalences that were proved, the level of confidence achieved by proving them in a proof assistant would be greater than an automatic verification by an SMT solver due to the smaller trusted code-base of proof assistants in relation to those of automatic theorem provers such as SMT solvers.

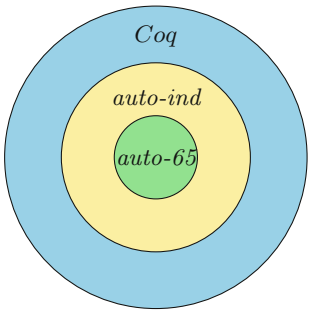


**Fig. 1.** The level of confidence achieved by the different approaches.

Figure 1 depicts the level of confidence achieved by the various approaches to verify invertibility equivalences. The smallest circle, labelled *auto-65*, represents the approach taken by [15], where invertibility equivalences were verified automatically up to 65 bits. While a step in the right direction, this approach is insufficient, because invertibility conditions are used for arbitrary bit-widths. The next circle, labeled *auto-ind*, depicts the approach of [17], which addresses the restrictions of auto-65 by providing bit-width independent proofs of the invertibility equivalences. However, both auto-65 and auto-ind provide proofs by SMT solvers, which are less trusted than ITPs. The largest circle (*Coq*) corresponds to work presented in the current paper which, while addressing the limitations of auto-65 via bit-width independent proofs, also provides stronger verification guarantees by proving the equivalences in an interactive theorem prover. Moreover, with this approach, we were able to prove equivalences that couldn't be fully verified (for arbitrary bit-widths) by either auto-65 or auto-ind.

## 4 The `BVList` Library

In this section, we describe the Coq library we use and the extensions we developed with the goal of formalizing and proving invertibility equivalences. Vari-

ous formalizations of bit-vectors in Coq exist. The internal Coq library of bit-vectors [9] is one, but it has only definitions and no lemmas. The Bedrock Bit Vectors Library [6] treats bit-vectors as words (machine integers). The SSRBit Library [5] represents bit-vectors as finite bit-sets in Coq and extracts them to OCaml machine integers. Our library is more suited to the SMT-LIB 2 bit-vectors, and includes operators that are not fully covered by any of the previously mentioned libraries. More recently, Shi et al. [22] developed a library called CoqQFBV that presents a bit-vector type as a sequence of Booleans, defines operators over it, and proves the correctness of these operations with respect to a (machine integer) semantics. [22] uses this library to define a bit-blasting algorithm in Coq, that is extracted into an OCaml program to perform certified bit-blasting. Since CoqQFBV covers the entire SMT-LIB 2 bit-vector signature, it would be a good alternative to ours in formalizing and proving invertibility conditions. Our library offers a rich set of lemmas over bit-vector operations that makes it suitable for proofs of invertibility conditions and other bit-vector properties. Bit-vectors have also been formalized in other proof assistants. Within the Isabelle/HOL framework, one can utilize the library developed by Beeren et al. [4] to align with SMT-LIB 2 bit-vector operations. Furthermore, Harrison [1] presents a formalization of finite-dimensional Euclidean space within HOL light, accompanied by an implementation of vectors.

### 4.1 `BVList` Without Extensions

`BVList` was developed for SMTCoq [10], a Coq plugin that enables Coq to dispatch proofs to external proof-producing solvers. While the library was only briefly mentioned in [10], here we provide more details.

The library adopts the little-endian notation for bit-vectors, following the internal representation of bit-vectors in SMT solvers such as cvc5, and corresponding to lists in Coq. This makes arithmetic operations easier to perform since the least significant bit of a bit-vector is the head of the Boolean list that represents it.

Another choice is how to formalize the bit-vector type. A dependently-typed definition is natural, since then the type of a bit-vector is parameterized by its length. However, such a representation leads to some difficulties in proofs. Dependent pattern-matching or case-analysis with dependent types is cumbersome and unduly complex (see, e.g., [23]), because of the complications brought by unification in Coq (which is inherently undecidable [24]). A simply-typed definition, on the other hand, does not provide such obstacles for proofs, but is less natural, as the length becomes external to the type. The `BVList` library defines for convenience both the dependently and the simply typed version of bit-vectors. It uses the Coq module system to separate them, and a functor that connects them, avoiding redundancy. The relationship between the two definitions is depicted in Fig. 2.

In `BVList`, a dependently-typed bit-vector is a record parameterized by its size $n$ and consisting of two fields: a Boolean list and a condition to ensure that the list has length $n$. This type, and the corresponding lemmas and properties over it, are encapsulated by the `BITVECTOR_LIST` module of type `BITVECTOR`. A

simply-typed or *raw* bit-vector representation is simply a Boolean list which, along with its associated operators and lemmas is specified by module signature `RAWBITVECTOR` and implemented in module `RAWBITVECTOR_LIST`. In other words, the interface of `BVList` offers dependently-typed bit-vectors, while the underlying operators are defined and proofs are performed using raw bit-vectors.

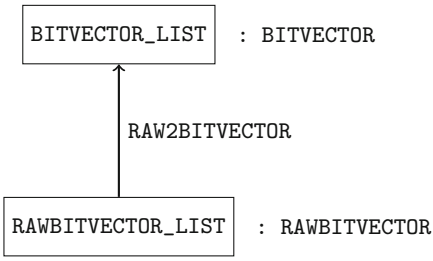


**Fig. 2.** Modular separation of `BVList`

A functor called `RAW2BITVECTOR` derives corresponding definitions and proofs over dependently-typed bit-vectors within the module for dependent-types, when it is applied to `RAWBITVECTOR_LIST`. The functor establishes a correspondence between the two theories so that one can first prove a bit-vector property in the context of the simply-typed theory and then map it to its corresponding dependently-typed one via the functor module. Otherwise put, users of the library can encode theorem statements more naturaly, and in a more expressive environment employing dependent types. For proofs, one can unlift them (by the functor) to the equivalent encodings with simple types, and prove them there.

### 4.2 Extending `BVList`

Out of the 13 bit-vector functions and 10 predicates contained in $\Sigma_1$, `BVList` had direct support for 10 functions and 6 predicates. The predicate symbols that were not directly supported were the weak inequalities $\leq_u$, $\geq_u$, $\leq_s$, $\geq_s$ and the unsupported function symbols were $\gg_a$, $\div$, and $\mathrm{mod}$. We extended `BVList` with the operator $\gg_a$ and the predicates $\leq_u$ and $\geq_u$ in order to support the corresponding invertibility conditions. Additionally, we redefined $\ll$ and $\gg$ in order to simplify the proofs of invertibility conditions over them.[1]

We focused on invertibility conditions for literals of the form $x \diamond s \bowtie t$ and $s \diamond x \bowtie t$, where $\diamond$ and $\bowtie$ are respectively function and predicate symbols in $\Sigma_0$. $\Sigma_0$ was chosen as a representative set because it is both expressive enough (in the sense that other operators can be easily translated to this fragment), and

[1] Both the extended library and the proofs of invertibility equivalences can be found at https://github.com/ekiciburak/bitvector/tree/frocos23.

feasible for proofs in Coq using the library. In particular, it was chosen as one that would require the minimal amount of changes to `BVList`. As a result, such literals, as well as their invertibility conditions, contain only operators supported by `BVList` (after its extension with $\gg_a$, $\leq_u$, and $\geq_u$). Supporting the full set of operators in $\Sigma_1$, both in the library and the proofs is left for future work.

```
Fixpoint ule_list_big_endian (x y : list bool) :=
  match x, y with
   | [ ], [ ] ⇒ true
   | [ ], _  ⇒ false
   | _, [ ] ⇒ false
   | xi:: x',  yi:: y' ⇒ ((eqb xi yi) && (ule_list_big_endian x' y'))
                        ||  ((negb xi) && yi)
  end.

Definition ule_list (x y: list bool) :=
  (ule_list_big_endian (rev x) (rev y)).

Definition bv_ule (a b : bitvector) :=
  if @size a =? @size b then
    ule_list a b
  else
    false.

Definition bv_ule n (bv1 bv2:bitvector n) : bool := M.bv_ule bv1 bv2.
```

**Fig. 3.** Definitions of $\leq_u$ in Coq.

In what follows, we describe our extensions to `BVList` with weak unsigned inequalities, alternative definitions for logical shifts, and the arithmetic right shift operator.

**Weak Unsigned Inequalities.** We added both weak inequalities for unsigned bit-vectors, $\leq_u$ and $\geq_u$. We illustrate this extension via that of the $\leq_u$ operator (the extension of $\geq_u$ is similar). The relevant Coq definitions are provided in Fig. 3. The top three definitions (including the fixpoint) cover the simply-typed representation, and the fourth, `bv_ule` is the dependently-typed representation that invokes the definition with the same name from module `M` of type `RAWBITVECTOR`. Like most other operators, $\leq_u$ (over raw bit-vectors) is defined over a few *layers*. The function `bv_ule`, at the highest layer, ensures that comparisons are between bit-vectors of the same size and then calls `ule_list`. Since we want to compare bit-vectors starting from their most significant bits and the input lists start instead with the least significant bits, `ule_list` first reverses the two lists. Then it calls `ule_list_big_endian`, which we consider to be at the lowest layer of the definition. This function does a lexicographic comparison of the two lists, starting from the most significant bits.

To see why the addition of $\leq_u$ to the library is useful, consider, for example, the following parametric lemma, stating that $\sim 0$ is the largest unsigned bit-vector of its type:

$$\forall x : \sigma_{[n]}.\ x \leq_u \sim 0 \tag{3}$$

Without an operator for the weak inequality, we would write it as:

$$\forall x : \sigma_{[n]}.\ x <_u \sim 0 \vee x = \sim 0 \tag{4}$$

```
1  Definition shl_one_bit (a: list bool) :=
2    match a with
3     | [ ]⇒ [ ]
4     | _ ⇒ false :: removelast a
5    end.
6
7  Fixpoint shl_n_bits (a: list bool) (n: nat) :=
8    match n with
9     | O ⇒ a
10    | S n' ⇒ shl_n_bits (shl_one_bit a) n'
11   end.
12
13 Definition shl_n_bits_a (a: list bool) (n: nat) :=
14   if (n <? length a)%nat then
15     mk_list_false n ++ firstn (length a -n) a
16   else
17     mk_list_false (length a).
18
19 Theorem bv_shl_eq: forall (a b : bitvector), bv_shl a b = bv_shl_a a b.
```

**Fig. 4.** Various definitions of $\ll$.

In such cases, since the definitions of $<_u$ and $=$ have a similar structure to that of $\leq_u$, we strip down the layers of $<_u$ and $=$ separately, whereas using $\leq_u$, we only do this once.

**Left and Right Logical Shifts.** We have redefined the shift operators $\ll$ and $\gg$ in `BVList`. Figure 4 shows both the original and new definitions of $\ll$. Those of $\gg$ are similar. Originally, $\ll$ was defined using the `shl_one_bit` and `shl_n_bits`. The function `shl_one_bit` shifts the bit-vector to the left by one bit and is called by `shl_n_bits` as many times as necessary. The new definition `shl_n_bits_a` uses `mk_list_false` which constructs the necessary list of 0 bits and appends (`++` in Coq) to it the bits to be shifted from the original bit-vector, which are retrieved using the `firstn` function, from the Coq standard library for lists. The `nat` type used in Fig. 4 is the Coq representation of Peano natural numbers that has `O` and `S` as its two constructors — as depicted in the cases rendered by pattern matching $n$ (lines 9-10). The theorem at the bottom of

Fig. 4 asserts the equivalence of the two representations, allowing us to switch between them, when needed. In the extended library, `bv_shl` defines the left shift operation using `shl_n_bits` whereas `bv_shl_a` does it using `shl_n_bits_a`. This new representation was useful in proving some of the invertibility equivalences over shift operators (see, e.g., Example 4 below).

**Arithmetic Right Shift.** Unlike logical shifts that were already defined in `BVList` and for which we have added alternative definitions, arithmetic right shift was not defined at all. We provided two alternative definitions for it, very similar to the definitions of logical shifts — `bv_ashr` and `bv_ashr_a`. Both definitions are conditional on the sign of the bit-vector (its most-significant bit). Apart from this detail, the definitions take the same approach taken by `shl_n_bits` and `shl_n_bits_a` from Fig. 4. Operator `bv_ashr` uses the definition of an independent shift and repeats it as many number of times as necessary, and `bv_ashr_a` uses either `mk_list_false` or `mk_list_true` to append the necessary number of sign bits to the shifted bits.

## 5 Proving Invertibility Equivalences in Coq

In this section we provide specific details about proving invertibility equivalences in Coq. We start by outlining the general approach for proving invertibility equivalences in Sect. 5.1. Then, Sect. 5.2 presents detailed examples of such proofs. Section 5.3 summarizes the results and impact of these proofs.

### 5.1 General Approach

The natural representation of bit-vectors in Coq is the dependently-typed representation, and therefore the invertibility equivalences are formulated using this representation. In keeping with the modular approach described in Sect. 4, however, proofs in this representation are composed of proofs over simply-typed bit-vectors, which are easier to reason about. Most of the work is on proving an equivalence over raw bit-vectors. Then, we derive the proof of the corresponding equivalence over dependently-typed bit-vectors using a smaller, boilerplate set of tactics. Since this derivation process is mostly the same across many equivalences, these tactics are a good candidate for automation in the future.

When proving an invertibility equivalence $IC[s,t] \Leftrightarrow \exists x.\ \ell[x,s,t]$, we first split it into two sub-goals: the left-to-right and right-to-left implications. For proving the left-to-right implication, since Coq implements a constructive logic, the only way to prove an existentially quantified formula is to construct the literal witnessing it. Thus, in addition to being able to prove the equivalence, a positive side-effect of our proofs are actual inverses for $x$ in literals of the form $\ell[x,s,t]$. In Niemetz et al. [16], these are called *conditional inverses*, as the fact that they are inverses is conditional on the correctness of the invertibility condition. There, such inverses were synthesized automatically for a subset of the literals. In each of our Coq proofs, such an inverse is found, even when the

proof is done by case-splitting. This provides a more general solution than the one in [16], which did not consider case-splitting.

*Example 2.* Consider the literal $s \gg_a x \geq_u t$. Its invertibility condition is $(s \geq_u \sim s) \vee (s \geq_u t)$. The left-to-right implication of the invertibility equivalence is:

$$\forall s, t : \sigma_{[n]}.\ (s \geq_u \sim s) \vee (s \geq_u t) \Rightarrow \exists x : \sigma_{[n]}.\ s \gg_a x \geq_u t$$

Here, case splitting is done on the disjunction in the invertibility condition. When $s \geq_u \sim s$ is true, the inverse for $x$ is the bit-vector constant that correspond to the length of the $s$, namely $n$; when $s \geq_u t$ is true, the inverse is 0. □

In addition to `BVList`, several proofs of invertibility equivalences benefited from CoqHammer [7], a plug-in that aims at extending the level of automation in Coq by combining machine learning and automated reasoning techniques in a similar fashion to what is done in by Sledgehammer [21] in Isabelle/HOL [18]. CoqHammer, when triggered on some Coq goal, (*i*) submits the goal together with potentially useful terms to external solvers/automated-provers, (*ii*) attempts to reconstruct returned proofs (if any) directly in the Coq tactic language Ltac [8], and (*iii*) outputs the set of tactics closing the goal in case of success. As we directly employ these tactics inside `BVList`, one does not need to install CoqHammer in order to build the library, although it would be beneficial for further extensions.

### 5.2 Detailed Examples

In this section we provide specific examples for proofs of invertibility equivalences. The first example illustrates the two-theories approach of the library.

*Example 3.* Consider the literal $s \gg_a x <_u t$. Its invertibility condition is $((s <_u t \vee \neg(s <_s 0)) \wedge t \neq 0)$. Figure 5 shows the proof of the following direction of the corresponding invertibility equivalence:

$$\forall s, t : \sigma_{[n]}.\ (\exists x : \sigma_{[n]}.\ s \gg_a x <_u t) \Rightarrow ((s <_u t \vee \neg(s <_s 0)) \wedge t \neq 0)$$

In the proof, lines 8–11 transform the dependent bit-vectors from the goal and the hypotheses into simply-typed bit-vectors. Then, lines 12–14 invoke the corresponding lemma for simply-typed bit-vectors (called `InvCond.bvashr_ult2_rtl`) along with some simplifications. □

Most of the effort in this project went into proving equivalences over raw bit-vectors, as the following example illustrates.

*Example 4.* Consider the literal $x \ll s >_u t$. Its invertibility condition is $(t <_u \sim 0 \ll s)$. The corresponding invertibility equivalence is:

$$\forall s, t : \sigma_{[n]}.\ (t <_u \sim 0 \ll s) \Leftrightarrow (\exists x : \sigma_{[n]}.\ x \ll s >_u t) \tag{5}$$

The left-to-right implication is easy to prove using $\sim 0$ itself as the witness of the existential proof goal and considering the symmetry between $>_u$ and $<_u$. The proof of the right-to-left implication relies on the following lemma:

$$\forall x, s : \sigma_{[n]}.\ (x \ll s) \leq_u (\sim 0 \ll s) \tag{6}$$

From the right side of the equivalence in Eq. (5), we get some skolem $x$ for which $x \ll s >_u t$ holds. Flipping the inequality, we have that $t <_u x \ll s$; using this, and transitivity over $<_u$ and $\leq_u$, the lemma given by Eq. (6) gives us the left side of the equivalence in Eq. (5).

As mentioned in Sect. 4, we have redefined the shift operators $\ll$ and $\gg$ in the library. This was instrumental, for example, in the proof of Eq. (6).

```
Theorem bvashr_ult2_rtl :
 forall (n : N), forall (s t : bitvector n),
 (exists (x : bitvector n), (bv_ult (bv_ashr_a s x) t = true)) ->
 (((bv_ult s t = true) ∨ (bv_slt s (zeros n)) = false) ∧
 (bv_eq t (zeros n)) = false).
Proof.
 intros n s t H.
 destruct H as ((x, Hx), H).
 destruct s as (s, Hs).
 destruct t as (t, Ht).
 unfold bv_ult, bv_slt, bv_ashr_a, bv_eq, bv in *. cbn in *.
 specialize (InvCond.bvashr_ult2_rtl n s t Hs Ht); intro STIC.
 rewrite Hs, Ht in STIC. apply STIC.
 now exists x.
Qed.
```

**Fig. 5.** A proof of one direction of the invertibility equivalence for $\gg_a$ and $<_u$ using dependent types.

The new definition uses `firstn` and `++`, over which many useful properties are already proven in the standard library. This benefits us in manual proofs, and in calls to CoqHammer, since the latter is able to use lemmas from the imported libraries to prove the goals that are given to it. Using this representation, proving Eq. (6) reduces to proving Lemmas `bv_ule_1_firstn` and `bv_ule_pre_append`, shown in Fig. 6. The proof of `bv_ule_pre_append` benefited from the property `app_comm_cons` from the standard list library of Coq, whereas `firstn_length_le` was useful in reducing the goal of `bv_ule_1_firstn` to the Coq equivalent of Eq. (3). The statements of the properties mentioned from the standard library are also shown in Fig. 6. □

Finally, we examine what was considered a challenge problem in the previous version of this work [11]. The next example details how we completed the proof.

*Example 5.* Consider the literal $(x \gg s) >_u t$. Its invertibility condition is $t <_u (\sim s \gg s)$. Now consider the following direction of the corresponding invertibility equivalence:

$$\forall s, t : \sigma_{[n]}.\ t <_u (\sim s \gg s) \Rightarrow \exists x : \sigma_{[n]}.\ (x \gg s) >_u t \tag{7}$$

Figure 7 contains the theorem stating the equivalence, and some lemmas used within its proof. A crucial step in the proof of the implication is to rewrite the definition of the right shift operator `bv_shr` to its alternate definition `bv_shr_a` (see Sect. 4.2). Unfolding the alternative definition leads to a case-analysis on the following condition:

$$\texttt{toNat}(s) < \texttt{len}(x)$$

where `toNat` casts a bit-vector to its natural number representation, and `len` returns the length of a bit-vector as a natural number.

```
1  Lemma bv_ule_1_firstn : forall (n : nat) (x : bitvector),
2   (n < length x)%nat ->
3   bv_ule (firstn n x) firstn n (mk_list_true (length x))) = true.
4
5  Lemma bv_ule_pre_append : forall (x y z : bitvector),
6   bv_ule x y  = true -> bv_ule (z ++ x) (z ++ y) = true.
7
8  Theorem app_comm_cons : forall (x y:list A) (a:A),
9   a ::  (x ++ y) = (a :: x) ++ y
10
11 Lemma firstn_length_le: forall l:list A, forall n:nat,
12  n <= length l -> length (firstn n l) = n.
```

**Fig. 6.** Examples of lemmas used in proofs of invertibility equivalences.

The challenge in the proof arises in the positive case of the condition, which reduces to a proof of `first_bits_zero` (see Fig. 7). `first_bits_zero` says that given $\texttt{toNat}(s) < \texttt{len}(s)$, the most-significant $\texttt{len}(s) - \texttt{toNat}(s)$ bits of $s$ are 0. As seen in Fig. 4, the second argument to the top-most layer of the shift (called from `bv_shl_eq`) is a bit-vector that specifies the number of times to shift the bit-vector in the first argument. This second argument is converted to a natural number by the abstract `toNat` function invoked above, the concrete definitions of which are specified in Fig. 7 as `list2nat_be_a` and `list2N`. At the same level of abstraction, we use `rev` for the list reversal function corresponding to the Coq function of the same name, and `firstn` also for its Coq namesake (`firstn` $n$ $l$ returns the $n$ most significant bits of $l$), so that `first_bits_zero` can be specified as follows:

$$\texttt{toNat}(s) < \texttt{len}(s) \Rightarrow \texttt{firstn}\ (\texttt{len}(s) - \texttt{toNat}(s))\ (\texttt{rev}(s)) = 0$$

The intuition behind its validity is that if the most-significant $\texttt{len}(s) - \texttt{toNat}(s)$ bits were not 0 then they would contribute to the value of $\texttt{toNat}(s)$, making it

greater than or equal to `len`($s$) and thus falsifying the condition. However, it is challenging to convert this intuition into a proof using induction over lists, as explained in what follows.

To prove `first_bits_zero`, we redefined `list2N` as a tail-recursive function `list2NTR`. This step was proven to be sound by a lemma of equivalence between the two definitions (`list2N_eq`). Since `list2N` is not tail recursive, it only begins computation at the end of the input list representing a bit-vector. Such a definition further complicates the proof of `first_bits_zero` when based on the typical induction principle over the structure of the Boolean list underlying the bit-vector `s`. This is because it does not easily reduce (via $\iota$-reduction for inductive definitions [19]), into a useful expression in the step case of the intended induction.

The advantage of tail recursion in this context is best illustrated by Fig. 8 where `x` is a Boolean variable and `xs` represents an arbitrary Boolean list. The

```
1   Theorem bvshr_ugt_ltr : forall (n : N), forall (s t : bitvector n),
2         (bv_ult t (bv_shr (bv_not s) s) = true) ->
3       (exists (x : bitvector n), bv_ugt (bv_shr x s) t = true).
4
5   Lemma first_bits_zero : forall (s : bitvector),
6    (N.to_nat (list2N s) < length s)%nat ->
7    firstn (length s - N.to_nat (list2N s)) (rev s) =
8    mk_list_false (length s -N.to_nat (list2N s)).
9
10  Lemma first_bits_zeroA : forall (s : bitvector),
11   (length s >= (list2NTR s))%nat ->
12   firstn (length s - (list2NTR s)) s =
13   mk_list_false (length s -(list2NTR s)).
14
15  Fixpoint list2N (a: list bool) :=
16   match a with
17    | [ ] ⇒ 0
18    | x ::   xs ⇒ if x then N.succ_double (list2N xs) else
19                    N.double (list2N xs)
20   end.
21
22  Definition list2nat_be_a (a: list bool) := N.to_nat (list2N a).
23
24  Fixpoint list2NR (a: list bool) (n: nat) :=
25   match a with
26    | [ ]      ⇒ n
27    | x ::  xs ⇒ if x then list2NR xs (2 * n + 1) else
28                  list2NR xs (2 * n)
29  end.
30
31  Definition list2NTR (a: list bool) := list2NR a 0.
32
33  Lemma list2N_eq: forall (s: bitvector),
34   list2NTR (rev s) = N.to_nat (list2N s).
```

**Fig. 7.** Invertibility equivalence for $\gg$ and $>_u$ and some lemmas used by its proof.

$$\frac{x\texttt{: bool} \quad xs\texttt{: list bool} \quad \texttt{IH: firstn } (\texttt{len}(xs) - \texttt{toNat}(xs))\ (\texttt{rev}(xs)) = 0}{\texttt{Goal: firstn } (\texttt{len}(xs) + 1 - \texttt{toNat}(x\ \texttt{::}\ xs))\ (\texttt{rev}(x\ \texttt{::}\ xs)) = 0} \quad (8)$$

$$\frac{x\texttt{: bool} \quad xs\texttt{: list bool} \quad \texttt{IH: firstn } (\texttt{len}(xs) - \texttt{toNatTR}(xs))\ (xs) = 0}{\texttt{Goal: firstn } (\texttt{len}(xs) + 1 - \texttt{toNatTR}(xs\ \texttt{++}\ [x]))\ (xs\ \texttt{++}\ [x]) = 0} \quad (9)$$

**Fig. 8.** Sub-goals generated in the proof of `first_bits_zero`. Note that 0 is a bit-vector constant of the appropriate length (list of `false`s).

derivation of the goal from the inductive hypothesis (`IH`) in derivation (8) from Fig. 8 is complicated in Coq because the functions `firstn` and `rev` are not well-matched with `list2N`, if not incompatible. For instance, observe that the in the inductive step (`Goal`), as the first argument to `firstn` increases, the number of bits fetched from the list increases towards the *right*. However, due to the little-endian notation of bit-vectors and the fact that the list `cons` function (`::`) can be seen as incrementing its argument list to its *left*, the `rev` function must be used to corrects the direction of increase of the second argument to `firstn`. Despite this correction, an induction over $s$ must deal with two structurally different lists.

In contrast, the tail-recursive definition of `list2NTR` hides the `rev` function. This is illustrated in derivation (9) in Fig. 8, where `toNatTR` corresponds to `list2NTR`. Furthermore, such an induction over lists using `append` (`++`) to the right, rather than `cons` to the left is possible thanks to the *reverse induction principle*[2]. Closing such a goal allowed us to prove the `list2NTR`-variant of `first_bits_zero`, specified as `first_bits_zeroA` in Fig. 7, and the proof of equivalence between the two definitions (`list2N_eq`) allowed us to use this in closing the original goal (7). □

### 5.3 Results

Table 2 summarizes the results of proving invertibility equivalences for invertibility conditions in the signature $\Sigma_0$. In the table, ✓ means that the invertibility equivalence was successfully verified in Coq but not in Niemetz et al. [17], and ✓ means the opposite; ✓✓ means that the invertibility equivalence was verified using both approaches. We successfully proved all invertibility equivalences over $=$ that are expressible in $\Sigma_0$, including 4 that were not proved in [17]. For the rest of the predicates, we focused only on the 8 invertibility equivalences that were not proved in [17], and succeeded in proving all of them.

Our work thus complements [17] in verifying all invertibility conditions in $\Sigma_0$ for arbitrary bit-widths, by proving all 12 equivalences that were previously unverified, and corroborating 7 others that were verified by SMT solvers. It also complements [15], which verified all invertibility conditions in $\Sigma_1$, but only up to bit-width of 65.

[2] see `rev_ind` in https://coq.inria.fr/library/Coq.Lists.List.html.

**Table 2.** Proved invertibility equivalences in $\Sigma_0$ where $\bowtie$ ranges over the given predicate symbols. ✓ means that the invertibility equivalence was successfully verified in Coq but not in [17], whereas ✓ means the opposite; ✓✓ means that the invertibility equivalence was verified using both approaches.

| $\ell[x]$ | $=$ | $\neq$ | $<_u$ | $>_u$ | $\leq_u$ | $\geq_u$ |
|---|---|---|---|---|---|---|
| $-x \bowtie t$ | ✓✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $\sim x \bowtie t$ | ✓✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $x \,\&\, s \bowtie t$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $x \mid s \bowtie t$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $x \ll s \bowtie t$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $s \ll x \bowtie t$ | ✓✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $x \gg s \bowtie t$ | ✓✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $s \gg x \bowtie t$ | ✓✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $x \gg_a s \bowtie t$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $s \gg_a x \bowtie t$ | ✓✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $x + s \bowtie t$ | ✓✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

## 6 Conclusion and Future Work

We have described our work on verifying bit-vector invertibility conditions in the Coq proof assistant, which required extending the `BVList` library in Coq. In addition to describing the library and our extensions to it, this paper presented details about the Coq proofs of the invertibility equivalences. These were done on a representative subset of the operators from the theory of bit-vectors that is well-supported by the extended library. We were able to prove in Coq all the equivalences that were left unproven in previous attempts for all bit-widths, and also to prove in Coq some equivalences that were proven automatically before, thus increasing confidence in their correctness.

The most immediate direction for future work is proving more of the invertibility equivalences supported by the bit-vector library. In addition, we plan to extend the library so that it supports the full syntax in which invertibility conditions are expressed, namely $\Sigma_1$. This will also increase the potential usage of the library for other applications. Another direction for future work is to extend the proofs for invertibility conditions where some of the bits are known. Such invertibility conditions were introduced by Niemetz and Preiner [14]. However, their formal verification for every bit-width is yet to be done.

**Acknowledgements.** This work was funded in part by NSF-BSF grant numbers 2110397 (NSF) and 2020704 (BSF), and ISF grant number 619/21.

## References


1. Harrison, J.: A HOL theory of euclidean space. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 114–129. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_8
2. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: A. Gupta & D. Kroening, editors: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
4. Beeren, J., et al.: Finite Machine Word Library. Archive of Formal Proofs. https://isa-afp.org/entries/WordLib.html Formal proof development (2016)
5. Blot, A., Dagand, P.É., Lawall, J.: Bit Sequences and Bit Sets Library. Available at https://github.com/pedagand/ssrbit
6. Chajed, T., et al.: Bedrock Bit Vectors Library. Available at https://github.com/mit-plv/bbv
7. Czajka, L., Kaliszyk, C.: Hammer for Coq: automation for dependent type theory. J. Autom. Reason. **61**(1-4), pp. 423–453 (2018). https://doi.org/10.1007/s10817-018-9458-4
8. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNAI, vol. 1955, pp. 85–95. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44404-1_7
9. Duprat, J.: Library Coq. Bool. Bvector. https://coq.inria.fr/library/Coq.Bool.Bvector.html
10. Ekici, B., et al.: SMTCoq: a plug-in for integrating SMT solvers into Coq. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 126–133. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_7
11. Ekici, B., Viswanathan, A., Zohar, Y., Barrett, C.W., Tinelli, C.: Verifying Bit-vector Invertibility Conditions in Coq (Extended Abstract). In Giselle Reis & Haniel Barbosa, editors: Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019. EPTCS **301**, pp. 18–26 (2019). https://doi.org/10.4204/EPTCS.301.4. Available at https://doi.org/10.4204/EPTCS.301.4
12. Herbert, B. Enderton (2001): Chapter TWO - First-Order Logic. In Herbert B. Enderton, editor: A Mathematical Introduction to Logic (Second Edition), second edition edition, Academic Press, Boston, pp. 67–181, https://doi.org/10.1016/B978-0-08-049646-7.50008-4
13. Gupta, A., Fisher, A.L.: Representation and symbolic manipulation of linearly inductive boolean functions. In: Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design, ICCAD '93, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 192–199 (1993). Available at http://dl.acm.org.stanford.idm.oclc.org/citation.cfm?id=259794.259827
14. Niemetz, A., Preiner, M.: Ternary Propagation-Based Local Search for more Bit-Precise Reasoning. In: FMCAD, IEEE, pp. 214–224 (2020)
15. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 236–255. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_16

16. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards satisfiability modulo parametric bit-vectors. J. Autom. Reason. **65**(7), 1001–1025 (2021). https://doi.org/10.1007/s10817-021-09598-9
17. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards bit-width-independent proofs in SMT Solvers. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 366–384. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_22
18. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): 5. the rules of the game. In: Isabelle/HOL. LNCS, vol. 2283, pp. 67–104. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9_5
19. Paulin-Mohring, C.: Inductive definitions in the system Coq rules and properties. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 328–345. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0037116
20. Paulin-Mohring, C.: Introduction to the Calculus of Inductive Constructions. In: Bruno Woltzenlogel Paleo & David Delahaye, editors: All about Proofs, Proofs for All, Studies in Logic (Mathematical logic and foundations) 55, College Publications. https://hal.inria.fr/hal-01094195 (2015)
21. Paulsson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In: Sutcliffe, G., Schulz, S., Ternovska, E., eds: The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011, EPiC Series in Computing 2, EasyChair, pp. 1–11, https://doi.org/10.29007/36dt. Available at https://doi.org/10.29007/36dt
22. Shi, X., Fu, Y.-F., Liu, J., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y.: CoqQFBV: a scalable certified SMT quantifier-free bit-vector solver. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 149–171. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_7
23. Sozeau, M.: Equations: A dependent pattern-matching compiler. In: Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP 2010), pp. 419–434 (2010). https://doi.org/10.1007/978-3-642-14052-5_29
24. Spies, S., Forster, Y.: Undecidability of higher-order unification formalised in Coq. In: Blanchette, J., Hritcu, C., eds.: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020, ACM, pp. 143–157, https://doi.org/10.1145/3372885.3373832. Available at https://doi.org/10.1145/3372885.3373832
25. The Coq development team (2019): The Coq Proof Assistant Reference Manual Version 8.9. Available at https://coq.inria.fr/distrib/current/refman/