Verifying the Conversion into CNF in Dafny

Viorel Iordache and Ştefan Ciobâcă ✉

Alexandru Ioan Cuza University, Iași, Romania
`iordacheviorel99,stefan.ciobaca@gmail.com`

Abstract. We present two computer-verified implementations of the CNF conversion for propositional logic. The two implementations are fully verified: functional correctness and termination is machine-checked using the Dafny language for both. The first approach is based on repeatedly applying a set of equivalences and is often presented in logic textbooks. The second approach is based on Tseitin's transformation and is more efficient. We present the main ideas behind our formalization and we discuss the main difficulties in verifying the two algorithms.

1 Introduction

Several computer-checked solvers for the boolean satisfiability problem have emerged relatively recently [18,22,5,9]. Most SAT solvers work with CNF-SAT, where the input formula is known to be in conjunctive normal form. This is not a limitation, since efficient algorithms to find the CNF of any formula are known. In this article, we address the problem of finding the CNF of a formula and we verify in the Dafny [15] language [1] two such algorithms.
The first approach that we verify is based on the standard textbook algorithm of applying a series of equivalences from left to right as long as possible. We work with the following nine equivalences:

$$(1)\ \varphi_1 \Leftrightarrow \varphi_2 \equiv (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1); \qquad (2)\ \varphi_2 \Rightarrow \varphi_1 \equiv \neg\varphi_1 \vee \varphi_2$$
$$(3)\ \varphi_1 \vee (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3);$$
$$(4)\ (\varphi_2 \wedge \varphi_3) \vee \varphi_1 \equiv (\varphi_2 \vee \varphi_1) \wedge (\varphi_3 \vee \varphi_1);$$
$$(5)\ \varphi_1 \vee (\varphi_2 \vee \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \vee \varphi_3; \qquad (6)\ \varphi_1 \wedge (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \wedge \varphi_2) \wedge \varphi_3;$$
$$(7)\ \neg(\varphi_1 \vee \varphi_2) \equiv \neg\varphi_1 \wedge \neg\varphi_2; \qquad (8)\ \neg(\varphi_1 \wedge \varphi_2) \equiv \neg\varphi_1 \vee \neg\varphi_2; \qquad (9)\ \neg\neg\varphi \equiv \varphi.$$

The first two equivalences remove implications and double implications, equivalences three and four distribute disjunctions over conjunctions, equivalences five and six associate parentheses in a standard form and the last three are used to push negations towards the leaves.
This approach has two advantages: it is simple and it does not introduce new variables. However, the disadvantage is that certain classes of formulae, such as $(x_1 \wedge x_1') \vee (x_2 \wedge x_2') \vee \ldots \vee (x_n \wedge x_n')$ $(n \geq 1)$, lead to exponentially large CNFs.

[1] We assume some familiarity with a system such as Dafny [15] or Why3 [6]. We give a very brief overview of Dafny in Appendix A for readers unfamiliar with Dafny.

First contribution. We verify in Dafny that an algorithm based on applying the nine equivalences above is functionally correct. The most difficult part is to prove termination, for which we use a carefully designed 5-tuple as a variant. To our knowledge, this is incidentally the first proof (paper or computer-checked) of termination of the nine rules. Indeed, in all logic textbooks that we surveyed, termination is only proved for a certain strategy (first applying rules 1 and 2, then finding a NNF using rules 7-9, etc.) of applying the rules. The interest into this is of theoretical interest, since other strategies (such as bringing the formula into NNF first) are easier to prove.
The second approach that we verify is a so-called definitional CNF [12] based on Tseitin's [2] transformation [24]. The idea is to create fresh boolean variables for each subformula. Each such fresh variable is constrained, by using carefully chosen clauses in the resulting CNF, to be equivalent to its associated subformula. This approach produces a CNF that is linear in the size of the given formula, with the theoretical disadvantage that the CNF is only equisatisfiable with the initial formula (not equivalent in general).
Second contribution. We verify an implementation of Tseitin's transformation in Dafny. The main difficulty is to find the right inductive invariant. There are also some technical difficulties with the verification: our implementation pushes the prover to its limits and requires carefully designed lemmas, helper predicates, and assertions in order to verify successfully. For this approach, termination is established by Dafny automatically, since the function is recursive on the formula ADT. To our knowledge, this is the first auto-active proof of Tseitin's transformation.
Paper structure. In Section 2 we present our verified implementation of the textbook-based CNF transformation and in Section 3 we present our verified implementation of Tseitin's transformation. In Section 4 we discuss related work and in Section 5 we conclude and discuss potential future work. There are two appendices: in Appendix A we give a very short overview of Dafny and in Appendix B we discuss the structure of our Dafny development. The Dafny development is available at `https://github.com/iordacheviorel/cnf-dafny`. We only present the most important parts of the development, which are necessary in order to understand our approach.

2 The Textbook Conversion into CNF

We describe our verified implementation of the textbook CNF transformation.

2.1 Data Structures

We represent boolean formulas as instances of the following algebraic data type:

```
datatype FormulaT = Var(val : int)  | Not(f1 : FormulaT)
| And(f1 : FormulaT, f2 : FormulaT) | Implies(f1 : FormulaT, f2 : FormulaT)
| Or(f1 : FormulaT, f2: FormulaT)   | DImplies(f1 :FormulaT, f2 : FormulaT)
```

[2] also spelled *Tseytin*.

We note that all standard logical connectives are available, that the connectives have a fixed arity, and that variables are represented by *non-negative integers*. The fact that variables are represented by non-negative integers is not encoded into the datatype; instead, we follow the standard Dafny practice of checking this as a precondition to all functions/methods computing with formulae by using a predicate that we call `validFormulaT`. We often require to know not merely that a formula is syntactically valid, but also to know that it contains at most n variables. For this purpose, we use a predicate `variablesUpTo`.We define `truthValue` to compute the truth value of a formula in a given assignment:

```
function method truthValue(f: FormulaT, assignment : seq<bool>) : bool
  decreases f; requires variablesUpTo(f, |assignment|);
{ match f {
    case Var(val) ⇒ assignment[val]
    case Not(f1) ⇒ ¬truthValue(f1, assignment)
    case And(f1,f2) ⇒ truthValue(f1,assignment) ∧ truthValue(f2,assignment)
    [...] } }
```

Truth assignments are represented as sequences of booleans (`seq<bool>`) that have sufficiently many elements to account for all variables in the formula, hence the `requires variablesUpTo(f, |assignment|)` precondition. As the truth value of a formula is used both in the specification and in the implementation of the algorithm, we declare it as a `function method`.
We define the predicate `equivalent` to check equivalence of two formulae:

```
predicate equivalent(f1 : FormulaT, f2 : FormulaT)
    requires validFormulaT(f1) ∧ validFormulaT(f2);
{ ∀ tau : seq<bool> • variablesUpTo(f1,|tau|) ∧ variablesUpTo(f2,|tau|)
      ⟹ truthValue(f1, tau) = truthValue(f2, tau) }
```

The predicate checks that the truth values of the two formulae are the same in any (sufficiently large) truth assignment.

2.2 Algorithm

We implement the CNF conversion algorithm using three methods:
• The method `applyAtTop` takes a formula and tries to apply one of the nine equivalence rules *at the root of the formula*. If it fails, the formula is returned unchanged.

```
method applyAtTop(f:FormulaT, ghost orsAbvLft:int, ghost andsAbvLft:int)
  returns (r : FormulaT) decreases f;
    requires orsAbvLft ≥ 0 ∧ andsAbvLft ≥ 0 ∧ validFormulaT(f);
    ensures validFormulaT(r) ∧ equivalent(f, r);
    ensures f = r ⟹ ¬f.Implies? ∧ f = r ⟹ ¬f.DImplies?;
    ensures r = f ∨ Utils.smaller(measure(r, orsAbvLft, andsAbvLft),
      measure(f, orsAbvLft, andsAbvLft));
{ match f {
    case DImplies(f1, f2) ⇒ { r := applyRule1(f, orsAbvLft, andsAbvLft); }
    case Implies(f1, f2)  ⇒ { r := applyRule2(f, orsAbvLft, andsAbvLft); }
    case Or(f1, f2) ⇒ { if (f2.And?) {
          r := applyRule3(f, orsAbvLft, andsAbvLft); } [...]
    } [...] } [...] }
```

We present the entire specification (pre- and post-conditions), but we skip some implementation details (replaced by `[...]`). The two ghost parameters are used in the termination proof, as discussed in Section 2.3. The first `ensures` clause

is used for functional correctness. The last `ensures` clauses are used to prove termination of the main algorithm. In particular, the function `measure` returns a tuple that acts as the variant of the main algorithm.
We also define the methods `applyRule1`, `applyRule2`, ..., `applyRule9`, which apply one of the nine rules. We present the first of these methods:

```
method applyRule1(f : FormulaT, ghost orsAbvLft : int,
    ghost andsAbvLft : int)
  returns (r : FormulaT) requires validFormulaT(f) ∧ f.DImplies?;
  requires orsAbvLft ≥ 0 ∧ andsAbvLft ≥ 0;
  ensures validFormulaT(r) ∧ equivalent(f, r);
  ensures weightOfAnds(r) ≤ weightOfAnds(f);
  ensures countDImplies(r) < countDImplies(f);
  ensures smaller(measure(r, orsAbvLft, andsAbvLft),
      measure(f, orsAbvLft, andsAbvLft));
{  var DImplies(f1, f2) := f;
   r := And(Implies(f1, f2), Implies(f2, f1));
   [...] }
```

Again, we show the entire specification, most of which is needed for the termination proof. The missing part in the implementation, denoted by `[...]`, are helper assertions and lemma calls that are required to prove the postconditions.
• The method `applyRule` takes a formula, traverses its tree in preorder, and calls `applyAtTop` to transform the first subformula where it is possible to do so at the root. Therefore, the method `applyRule` makes exactly one effective call to `applyAtTop`. We present the entire specification and the implementation of one of the cases for `applyRule`:

```
method applyRule(f : FormulaT, ghost orsAbvLft : int, ghost andsAbvLft : int)
  returns (r : FormulaT) decreases f;
  requires validFormulaT(f) ∧ orsAbvLft ≥ 0 ∧ andsAbvLft ≥ 0;
  ensures validFormulaT(r) ∧ equivalent(f, r);
  ensures r = f ∨ Utils.smaller(measure(r,orsAbvLft ,andsAbvLft),
    measure(f,orsAbvLft ,andsAbvLft));
{ var res : FormulaT := applyAtTop(f, orsAbvLft, andsAbvLft);
  if (res ≠ f) { return res; } else if (f.Or?) {
    var f1_step := applyRule(f.f1, orsAbvLft , andsAbvLft);
    if (f.f1 = f1_step) {
      var f2_step := applyRule(f.f2, orsAbvLft + 1, andsAbvLft);
      assert equivalent(f.f2, f2_step);
      assert equivalent(Or(f.f1, f.f2), Or(f.f1, f2_step));
      res := Or(f.f1, f2_step);
      if (weightOfAnds(f2_step) < weightOfAnds(f.f2)) {
        Rule3Or(f.f1, f.f2, f2_step); }
      return res;
    } else { [...] }
  } else if (f.And?) { [...] } else if (f.Not?) { [...] } }
```

Note again that the two ghost parameters are only used to help prove termination of the main algorithm and that the pre- and post-conditions are very similar to `applyRule`. Also note that there are no cases for `f.Implies?` and `f.DImplies?` since in these two cases `applyAtTop` is forced to make progress. The lemma `Rule3Or` is used to propagate a termination variant upwards in the tree of the formula and is explained in Section 2.3.
• The main method in the algorithm is `convertToCNF`. It takes a formula and calls `applyRule` on it in a recursive loop until there are no more changes.

```
method convertToCNF(f : FormulaT) returns (r : FormulaT)
  decreases weightOfAnds(f);/*3,4,7,8,9*/ decreases countDImplies(f);/*1*/
  decreases countImplies(f);/*2*/         decreases countOrPairs(f,0);/*5*/
```

```
    decreases countAndPairs(f, 0);/*6*/     requires validFormulaT(f);
    ensures validFormulaT(r) ∧ equivalent(f, r);
  { var res := applyRule(f, 0, 0); assert equivalent(f, res);
    if(res ≠ f) { r := convertToCNF(res);
                  assert equivalent(res, r); [...]
    } else { r := res; } }
```

The main difficulty here is to prove the termination of this fixed-point method. For this purpose, we use as a variant a 5-tuple `measure`, whose definition we unfold in the five `decreases` clauses of `convertToCNF`. The numbers in the comments represent the equivalences among the set of nine that ensure a strict decrease of the particular element of the tuple.

2.3 Proof of Termination

In this section, we discuss in more depth the proof of termination which *seems* intuitively easy: (I1) it *seems* that the first two equivalences strictly decrease the number of double implications and implications, respectively; (I2) it *seems* that equivalences three and four strictly decrease the number of disjunctions that sit above conjunctions in the tree of the formula; (I3) it *seems* that rule five (resp. six) strictly decrease the number of `or`s just above and to the left of another `or` (resp. `and`); (I4) it *seems* that rules 7-9 strictly decrease the number of negations above conjunctions and disjunctions. The above intuition works when using a particular strategy to compute a CNF: first, remove double implications; secondly, remove implications; thirdly, compute the NNF, etc.
Difficulties. Unfortunately, all of the above intuition breaks when the rules can be applied in any order. There are two main difficulties with the variant candidates above: (D1) Equivalences one, three, and four are not right-linear; they might duplicate subformulae. Therefore, the variants suggested by intuitions I1, I2, I3, I4 (e.g., number of double implications) might actually increase when applying one of these equivalences. The apparent solution of counting the number of distinct subformulae rooted in a double implication instead of the number of double implications does not work either: after the initial duplication, the two subformulae might be transformed in different ways. (D2) The variants suggested by intuitions I1-I4 do not in general extend homomorphically to the root of a formula when the transformation is performed in a proper subformula. Take for example the number of `and`s directly above and to the left of another `and` node (intuition I3). If we apply rule three in a context of the form $\varphi \wedge \square$, we obtain $\varphi \wedge ((\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3))$ from $\varphi \wedge (\varphi_1 \vee (\varphi_2 \wedge \varphi_3))$ and therefore our variant candidate actually increases at the root.
The main variant. In order to prove termination, we rely instead on a numerical interpretation of formulae that we call `weightOfAnds`. It decreases strictly in equivalences 3, 4, 7, 8, and 9 and it does not increase in the other equivalences.

```
  function weightOfAnds(f : FormulaT) : (res : int)
    decreases f; ensures res ≥ 2;
  { match f {
      case Var(val)        ⇒ 2
      case Not(f1)         ⇒ pow(2, weightOfAnds(f1))
      case And(f1, f2)     ⇒ weightOfAnds(f1) + weightOfAnds(f2) + 1
      case Or(f1, f2)      ⇒ weightOfAnds(f1) * weightOfAnds(f2)
      case Implies(f1, f2) ⇒ pow(2, weightOfAnds(f1)) * weightOfAnds(f2)
      case DImplies(f1,f2) ⇒ (pow(2,weightOfAnds(f1)) * weightOfAnds(f2)) +
                    (pow(2,weightOfAnds(f2)) * weightOfAnds(f1)) + 1 } }
```

Intuitively (although we admit this is not a perfect intuition), this function computes the number of conjunctions that a formula might have when brought into CNF, hence the `+ 1` in the `And(f1, f2)` case. For disjunctions, intuitively the `Or` needs to be distributed over all `And`s and therefore we use multiplication. For negation, the number might increase exponentially (negation "turns" `Or`s into `And`s and vice-versa). The `pow` function is not builtin; it performs exponentiation and is defined in the Dafny development along with several helper lemmas. For technical reasons, for leaves the counting starts at 2 (`case Var(val) => 2`).
The cases for implication and double implication are handled as if an implication $\varphi_1 \Rightarrow \varphi_2$ is just syntactic sugar for $\neg\varphi_1 \vee \varphi_2$ and therefore applying equivalences 1 and 2 trivially does not change the value of `weightOfAnds`. Equivalences 5 and 6 also do not change the value of `weightOfAnds`, as the numerical interpretations of $\vee$ and of $\wedge$ are associative. This numerical interpretation is homomorphic, and therefore difficulty D2 is handled.
Secondary variants. The main variant establishes termination of rules 3, 4, 7, 8, and 9. To establish termination of the entire system, we require 4 more secondary variants. The second and third elements of the tuple (`countDImplies(f)` and `countImplies(f)`) unsurprisingly count the number of double implications and implications in a formula, respectively. These decrease strictly when applying equivalences 1 and 2, respectively.
More interestingly, we discuss the last two components: `countOrPairs(f, 0)` and `countAndPairs(f, 0)`. These are used to establish termination of the rules for associating parentheses to the left (equivalences 5 and 6). As explained in difficulty D2 above, simply counting the number of `And` nodes directly above and to the left of another `And` node does not work, since the inequality required of such a variant does not propagate from a subformula towards the root. Therefore, we count instead the number of pairs of nodes labeled `And` such that one is a (possibly indirect) ancestor of the other towards the left. We display `countOrPairs`, as the other function is similar:

```
function countOrPairs(f : FormulaT, orsAbvLft : int) : (res : int)
  decreases f; requires orsAbvLft ≥ 0; ensures res ≥ 0;
{ match f {
    case Or(f11,f12) ⇒ countOrPairs(f11, orsAbvLft) +
      countOrPairs(f12, orsAbvLft + 1) + orsAbvLft // note the ‘‘+ 1’’
    case Var(val) ⇒ 0
    case And(f11,f12) ⇒ countOrPairs(f11, orsAbvLft) +
      countOrPairs(f12, orsAbvLft)
    [...] } }
```

The helper parameter represents the number of `Or`s above and to the left of the current subformula and should therefore be 0 in the initial call. This variant propagates as required.
In order to implement this variant, we require to keep track in the `applyRule` method of the number of `Or`s and `And`s that are possibly indirect ancestors towards the left of the current subformula. These numbers account for the two ghost parameters `orsAbvLft` and `andsAbvLft` of the `applyRule` and `applyAtTop`

methods mentioned above and left for the current subsection. Initially, when `applyRule` is called in `convertToCNF`, both are initialized to 0.
The first three components of the variant do not commute, but the last two could be interchanged without affecting the termination proof.

3 The Tseitin Conversion

We discuss our formalization of the Tseitin conversion into CNF [24], also called definitional CNF [12]. We first briefly recall the Tseitin transformation via a short example.

Example 1. Consider the formula $\varphi = \neg x_1 \vee x_2$. Choose fresh variables y_1, y_2 for the two subformulae $\neg x_1$ and $\neg x_1 \vee x_2$, respectively. Add to the resulting CNF clauses that encode that each of the two variables is equivalent to the corresponding subformula: 1. for $y_1 \equiv \neg x_1$, add the clauses $y_1 \vee x_1, \neg y_1 \vee \neg x_1$; 2. for $y_2 \equiv y_1 \vee x_2$, add the clauses: $\neg y_1 \vee y_2, \neg x_2 \vee y_2, \neg y_2 \vee y_1 \vee x_2$.
Finally, add a clause consisting of a single literal: the variable corresponding to the initial formula [3]. The final result is: $(y_1 \vee x_1) \wedge (\neg y_1 \vee \neg x_1) \wedge (\neg y_1 \vee y_2) \wedge (\neg x_2 \vee y_2) \wedge (\neg y_2 \vee y_1 \vee x_2) \wedge y_2$. The result of applying our implementation of Tseitin's algorithm on the above formula is the sequence of sequences $[[3,1],[-3,-1],[-3,4],[-2,4],[-4,3,2],[4]]$, where the numbers 1, 2, 3, and 4 represent the variables x_1, x_2, y_1, and y_2, respectively.

The guarantee offered by this transformation is that the resulting formula is equisatisfiable to the initial formula, as opposed to equivalent for the textbook transformation discussed in Section 2. The advantage is that the size is at most O(1) times bigger than the initial formula.

3.1 Data Structures

In verifying the definitional CNF transformation, we use the same data type `FormulaT` for the input formula as in the previous formalization. However, for the output formula, we choose to represent literals as integers, clauses as sequences of integers and CNF formulae as sequences of clauses, similar to the well-known DIMACS format:

```
predicate method validLiteral(lit : int) { lit ≤ -1 ∨ lit ≥ 1 }
predicate validClause(clause : seq<int>) {
  ∀ lit • lit in clause ⟹ validLiteral(lit) }
predicate validCnfFormula(f : seq<seq<int> >) {
  ∀ clause : seq<int> • clause in f ⟹ validClause(clause) }
```

Note that `valid` in the predicates above reflects standard Dafny use rather than semantical validity in the logical sense. As variables are represented by non-negative integers, we consistently use the following function (methods) to convert between variables and literals:

[3] Tseitin [24] proposes to add the negation of this literal, the initial formula being valid iff the resulting formula is unsatisfiable; modern treatments diverge [12].

```
predicate validVariable(v : int) { v ≥ 0 }
function method posVarToLit(v : int) : int
  requires validVariable(v);
  ensures posVarToLit(v) ≥ 1 ∧ validLiteral(posVarToLit(v));
{ v + 1 }
function method negVarToLit(v : int) : int
  requires validVariable(v);
  ensures negVarToLit(v) ≤ -1 ∧ validLiteral(negVarToLit(v));
{ (-v) - 1 }
function method litToVar(l : int) : int
  requires validLiteral(l);
{ if (l ≤ -1) then (-l) - 1 else l - 1 }
```

We represent assignments as sequences of booleans, just like in the textbook transformation. To compute truth values, we use the following predicates:

```
predicate truthValueLiteral(lit : int, tau : seq<bool>)
    requires validLiteral(lit) ∧ variablesUpToLiteral(lit, |tau|);
{ if lit < 0 then ¬tau[litToVar(lit)] else tau[litToVar(lit)] }
predicate truthValueClause(clause : seq<int>, tau : seq<bool>)
  requires validClause(clause) ∧ variablesUpToClause(clause, |tau|);
{ ∃ lit • lit in clause ∧ truthValueLiteral(lit, tau) }
predicate truthValueCnfFormula(rf : seq<seq<int⟩, tau : seq<bool>)
  requires validCnfFormula(rf) ∧ variablesUpToCnfFormula(rf, |tau|);
{ ∀ clause | clause in rf • truthValueClause(clause, tau) }
```

The predicates `variablesUpTo*` (with $* \in \{$`CnfFormula`, `Clause`, `Literal`$\}$), check that the assignment is sufficiently large to account for all variables occurring in the formulae. We consistently use the following convention: program variables such as `f`, `f1`, `f2` stand for formulae of type `FormulaT` and program variables such `rf`, `rf1` stand for *resulting formulae* of type `seq<seq<int> >`. As the guarantee of the Tseitin transformation is equisatisfiability between the initial formula and the resulting formula, we model this by using the following predicates:

```
predicate satisfiable(f : FormulaT) requires validFormulaT(f);
{ ∃ tau | |tau| = maxVar(f) • truthValue(f, tau) }
predicate satisfiableCnfFormula(rf: seq<seq<int> >) requires validCnf[...](rf);
{ ∃ tau | |tau| = maxVarCnfFormula(rf) • truthValueCnfFormula(rf, tau) }
predicate equiSatisfiable(f : FormulaT, rf : seq<seq<int> >)
    requires validFormulaT(f); requires validCnfFormula(rf);
{ satisfiable(f) ⟺ satisfiableCnfFormula(rf) }
```

The function methods `maxVar*` compute the maximum natural number that represents a variable inside the formulae, plus one. Therefore an assignment of size `maxVar*` is sufficiently large to compute the truth value.

3.2 The Algorithm

The entry point to the algorithm is the method `tseitin`.

```
method tseitin(f : FormulaT) returns (result : seq<seq<int> >)
  requires validFormulaT(f);
  ensures validCnfFormula(result) ∧ equiSatisfiable(f, result);
{ var n := maxVar(f); var v : int; var end : int; var rf : seq<seq<int> >;
  rf, v, end := tseitinCnf(f, n, n);
  result := rf + [[posVarToLit(v)]]; [...] }
```

The guarantee is that the resulting CNF formula is equisatisfiable to the initial one. The main work is performed by the method `tseitinCnf`, which traverses the input formula recursively and adds the right clauses to the result:

```
method tseitinCnf(f : FormulaT, n : int, start : int)
  returns (rf : seq<seq<int> >, v : int, end : int)
  requires variablesUpTo(f, n) ∧ start ≥ n ≥ 0;
  ensures valid(f, rf, v, n, start, end);
  ensures tseitinSameValue(f, rf, v, n, start, end);
  ensures tseitinCanExtend(f, rf, v, n, start, end);
{ match f {
    case Or(f1, f2)        ⇒ {
      var rf1 : seq<seq<int> >; var rf2 : seq<seq<int> >;
      var v1 : int; var v2 : int;
      var mid : int;
      rf1, v1, mid := tseitinCnf(f1, n, start);
      rf2, v2, v := tseitinCnf(f2, n, mid);
      end := v + 1;
      rf := rf1 + rf2 + orClauses(v1, v2, v);
      proveCanExtendOr(f1,rf1,v1, f2, rf2, v2, n, start, mid, v, end, rf);
      proveSameValueOr(f1,rf1,v1, f2, rf2, v2, n, start, mid, v, end, rf);
    }
    case And(f1, f2)      ⇒ [...]  case Implies(f1, f2)  ⇒ [...]
    case DImplies(f1, f2) ⇒ [...]  case Not(f1) ⇒ [...]
    case Var(val) ⇒ [...]
} }
```

The method `tseitinCnf` takes as input: 1. a formula `f` to transform into CNF, which might be a subformula of the initial formula given to `tseitin`; 2. a natural number `n` with the meaning that all variables in the initial formula are between 0 and $n-1$; 3. a natural number `start`, with the meaning that the variables `start`, `start + 1`, `start + 2`, ... are not used and can be safely used as fresh variables by the method. Variables between `n` (inclusively) and `start` (exclusively) might have been used for some other subformulae.
The method `tseitinCnf` returns as output: 1. A set of clauses `rf` encoding that the freshly chosen variables are equivalent to the corresponding subformulae; 2. A variable `v` that corresponds to the input formula `f`; 3. A number `end` with the meaning that the recursive call used fresh variables between `start` (inclusively) and `end` (exclusively) and therefore the variables `end`, `end + 1`, `end + 2`, ... can be used safely as fresh after the call is finished. The predicate `valid` is used to account for the validity of the entire state of the algorithm:

```
predicate valid(f : FormulaT, rf : seq<seq<int> >, v : int,
  n : int, start : int, end : int)
{ 0 ≤ n ≤ start ≤ end ∧ variablesUpTo(f, n) ∧ validCnfFormula(rf) ∧
  validVariable(v) ∧ variableInInterval(v, n, start, end) ∧
  variablesInInterval(rf, n, start, end) }
```

The predicate `variablesInInterval(rf, n, start, end)` checks that `rf` uses only the initial variables (between 0 and $n-1$) and fresh variables between `start` (inclusively) and `end` (exclusively).
We discuss in more detail the implementation of the `Or` case in `tseitinCnf` presented above. Note how the recursive call on `f1` uses fresh variables between `start` (inclusively) and `mid` (exclusively), while the recursive call on `f2` uses fresh variables between `mid` (inclusively) and `v` (exclusively). The variable `v` is therefore used as the fresh variable corresponding to the entire formula `f` = `Or(f1, f2)`. The final set of clauses is then the union of `rf1` (set of clauses corresponding to `f1`), `rf2` (set of clauses corresponding to `f2`) and `orClauses(v1, v2, v)`, which encodes that `v` should be equivalent to `Or(v1, v2)`:

```
function method orClauses(v1 : int, v2 : int, v : int) : seq<seq<int> >
```

```
  requires validVariable(v1) ∧ validVariable(v2) ∧ validVariable(v);
{ [[negVarToLit(v), posVarToLit(v1), posVarToLit(v2)],
    [negVarToLit(v1), posVarToLit(v)], [negVarToLit(v2), posVarToLit(v)]] }
```

3.3 The Proof

The main difficulty in verifying the algorithm is coming up with the right invariants. Assuming that `tseitinCnf(f, n, start)` returns `rf, v, end`, we find that the following two invariants explain the functional correctness of the algorithm: 1. any truth assignment `tau` to the initial `n` variables can be extended (uniquely) to a truth assignment `tau'` that makes `rf` true and such that the value of `f` in `tau` is the same as the value of `v` in `tau'`; 2. vice-versa, any truth assignment to all `end` variables that makes `rf` true also makes `v` and `f` have the same truth value. We formalize the two invariants above in the predicates `tseitinCanExtend` and `tseitinSameValue`:

```
predicate tseitinCanExtend(f : FormulaT, rf : seq<seq<int> >,
    v : int, n : int, start : int, end : int)
  requires valid(f, rf, v, n, start, end);
{ ∀ tau : seq<bool> | |tau| = n • canExtend(tau,f,rf,v,n,start,end) }

predicate canExtend(tau : seq<bool>, f : FormulaT, rf : seq<seq<int> >,
    v : int, n : int, start : int, end : int)
  requires |tau| = n ∧ valid(f, rf, v, n, start, end);
{ ∃ tau' : seq<bool> | tau ≤ tau' ∧ |tau'| = end •
    truthValueCnfFormula(rf, tau') ∧ truthValue(f, tau) =
    truthValueLiteral(posVarToLit(v), tau') }

predicate tseitinSameValue(f : FormulaT, rf : seq<seq<int> >,
    v : int, n : int, start : int, end : int)
  requires valid(f, rf, v, n, start, end);
{ ∀ tau : seq<bool> | |tau| ≥ end ∧ truthValueCnfFormula(rf, tau) •
    [...] truthValueLiteral(posVarToLit(v), tau) = truthValue(f, tau) }
```

We find that because `tseitinCanExtend` is of the form $\forall\_.\exists\_.\_$ (nested quantifiers), it is useful for verification performance to give a name to the $\exists\_.\_$ part, hence the predicate `canExtend`. Dafny cannot prove the two predicates automatically, and therefore we design helper lemma for each of the two invariants and for each of the cases (`Or`, `And`, `Not`, . . .).The main idea behind these proof is to combine two assignments `tau1` and `tau2`, which necessarily agree on the first n variables (the variables in the initial formula), into a single assignment `tau'`. This is possible since the *interesting* assignments in `tau1` range from `start` to `mid` and the *interesting* assignments in `tau2` range from `mid` to `v`; that is they are disjoint. We ellide the computer-checked proof for space reasons.

4 Related Work

The work closest to ours is by Barroso et al. [3], who verify a CNF transformation for propositional logic in the Why3 [6] verification platform. They use the textbook approach, but they rely on a particular strategy (first, remove implication, then: compute the negation normal form, etc.) This makes their proof much simpler, especially w.r.t. termination. One theoretical difference is that they model truth assignments as functions from variables to truth values and therefore their specification is closer to the mathematical treatment of logic (we instead model truth assignment as finite sequences, but we ensure they are sufficiently large for the context in which they are used). Barroso et al. emphasize the verification of continuation-passing style of the CNF transformation, which is out of the scope of our paper.

Michaelis and Nipkow [20] mechanize and prove Tseitin's transformation in Isabelle/HOL as part of the formalization [21] of a series of propositional proof systems. The implementation is functional, based on first generating fresh names for all distinct subformulae and then adding the corresponding clauses for each internal node of the input formula. The fact that the fresh names are generated at the very beginning seems to make the proof simpler. As the emphasis is placed on metatheoretical considerations, efficiency is not the main concern. In our Dafny approach, the implementation is more efficient and is compositional: each subformula is recursively translated into a set of clauses.

Gäher and Kunze [11] implement and verify Tsetin's transformation in the Coq proof assistant as part of the proof of the Cook-Levin theorem. The algorithm is implemented as a fixpoint (terminating, pure, recursive function) in the functional language of the Coq proof assistant. The function is very similar to our implementation: it takes a subformula and a natural starting from which fresh identifiers can be chosen. It returns the set of clauses for the subformula, the new variable associated to the subformula and a new number to be used for freshness. The inductive invariant `tseytin_formula_repr` used for the proof is also very similar to what we have independently found. Since the implementations are in very different proof environments, isolating Tseitin's transformation in both developments and performing a more detailed comparison could be used to understand the pros and cons of the two proof assistants (Dafny and Coq).

Verified transformation into CNF should be a first step in verified SAT solvers that take as input arbitrary formulae. The SAT solver `versat` [22] was implemented and verified in the Guru programming language using dependent types. The solver is verified to be sound: if it produces an `UNSAT` answer, then the input formula truly is unsatisfiable. Blanchette and others [5] present a certified SAT solving framework verified in the Isabelle/HOL proof assistant. The proof effort is part of the *Isabelle Formalization of Logic* project. The framework is based on refinement: at the highest level sit several calculi like CDCL and DPLL, which are formally proved. Depending on the strategy, the calculi are also shown to be terminating. Another SAT solver verified in Isabelle/HOL is by Marić [18]. In contrast to previous formalization, the verification methodology is not based on refinement. Instead, the Hoare triples associated to the solver pseudo-code are verified in Isabelle/HOL. In subsequent work [19], Marić and Janičić prove in Isabelle the functional correctness of a SAT solver represented as an abstract transition system. Andrici and Ciobâcă [2,1] verify an implementation of DPLL in Dafny. Another formalization of a SAT solver (extended with linear arithmetic) is by Lescuyer [17], who verifies a DPLL-based decision procedure for propositional logic in Coq and exposes it as a reflexive tactic. Finally, a decision procedure based on DPLL is also verified by Shankar and Vaucher [23] in the PVS system. For the proof, they rely on subtyping and dependent types. Berger et al. have used the Minlog proof assistant to extract a certified SAT solver [4]. **None of the verified SAT solvers described above perform a CNF conversion**, with the exception of the reflexive procedure by Lescuyer [17]. Lescuyer notes that implementing Tseitin's procedure in Coq proved to be *much more challenging* and therefore implements a lazy CNF transformation.

5 Discussion

Compiling (including verification time) the entire development (the two CNF transformations) takes about one minute on a standard laptop. The following table contains a summary of the entire development in numbers.

Lines of code	2440	Methods	32
Preconditions	318	Postconditions	176
Predicates	26	Functions	26
Assertions	227	Lemmas	51
Ghost variables	31	Verification time	∼1 min

We find that Dafny can be used successfully to complete this case study. However, the degree of proof automation is small and most of the interesting verified proofs require significant assistance from the user in the form of helper assertions and lemmas. Additionally, we found that the verified proofs of the Tseitin algorithm push Dafny to the edge, in the sense that a particular organization of the proofs is required in order for the development to verify in reasonable time. To this purpose, we propose the following *verification patterns* that help achieve good verification performance and that are portable to other Dafny developments:
(1) Do not use nested quantifiers. Instead, whenever a formula like $\forall \_.\exists \_.\_$ occurs, create a predicate $Q \equiv \exists \_.\_$ and use $\forall \_.Q$ instead of the initial formula. We use this verification pattern in the verification of the Tseitin transformation in the context of the `tseitinCanExtend` predicate. This transformation could be automated and could serve as an improvement in deductive verifiers, but further investigation into its merits on more case studies should be performed first.
(2) Do *not* inline even simple predicates. Make sure inlining is consistent. For example, we prefer to use the following trivial predicate:

```
predicate validVariable(v : int) { v ≥ 0 }
```

instead of inlining it. Additionally, consistency is required: mixing `v >= 0` and `validVariable(v)` will generally result in a potential performance degradation in verification (e.g., mixing `v >= 0` and `validVariable(v)` in one file results in a verification time increases by approx. 16% in our project).
(3) Consistently add post-conditions, even if the they are trivial. For example, if we remove the two post-conditions in the following function method:

```
function method negVarToLit(v : int) : int    requires validVariable(v);
  ensures negVarToLit(v) ≤ -1 ∧ validLiteral(negVarToLit(v));
{ (-v) - 1 }
```

our development still verifies, but takes approx. 50% more time to do so (approx. 1m30s instead of approx. 1m). On more complex functions, this can be the difference between verification succeeding and failing (for no apparent reason).

Incidentally, the above patterns do not only help Dafny verify the development faster, but also often clarify invariants for the programmer by forcing them to consistently give names to certain recurring formulae and enabling them to essentially create a mini-DSL for proofs.

Our case study also suggests a few areas where Dafny and other similar verifiers could be improved. For example, numerical functions such as `pow` (exponentiation) could be built in, possibly with some associated helper lemmas. Termination measures could be improved, both in allowed syntax (e.g., allowing `decrease userdefinedfunction(...)`), but also in allowing other well-founded orders such as multiset orderings (although we have finally not needed such orders in our development). Another area that could be improved is predictable verification performance. We find that, especially when not following the patterns described above, performance is very difficult to predict.

Implementation choices. For Tseitin's algorithm, we use a different representation for the input formula (an ADT) and the output formula (a set of clauses). Not only is this representation of the output the most natural for implementing the algorithm, but it is also what a SAT solver takes as input. Therefore, it allows in principle to easily combine our CNF transformation with a SAT solver. It would be easy to convert the set of clauses into the ADT representation and prove equivalence of the two, or to directly work with ADTs as part of the transformation. For the textbook transformation, the output and the input have the same representation (ADTs). However, we do not prove explicitly that the result is in CNF (it follows implicitly from the fact that none of the nine equivalences can be applied anymore). Proving this explicitly would require to first specify what is means for a formula to be in CNF; it would be an interesting exercise to prove this and to extract the set of clauses from the CNF.

Future work. Our case study opens several directions for future work. *Possible improvements.* The formulae could be represented as DAGs instead of trees. This could speed up both algorithms; more interestingly, this might simplify the termination proofs for the textbook algorithm as discussed in Difficulty D1 on Page 5. Several optimizations to Tseitin's algorithm [12,8] could also be implemented and verified. For a high-performance conversion, literals and variables should be represented as machine integers, which would require proving bounds throughout the code. *Theoretical extensions.* It would be interesting to find improved (possibly tight) bounds on the termination measure for the first algorithm. Additionally, it would be useful to bridge the distance between *truth assignment* as defined in the Dafny development and the usual notion of *truth assignment* in logic: our assignments are finite (with sufficiently many elements to account for all propositional variables in the context where they are used), while *truth assignments* are usually countably infinite. *Verification improvements.* It would be useful to further simplify the variant used to prove termination for the first algorithm. As Dafny supports a limited form of refinement types [10] (only numerical types can be refined by a logical constraint), it might be useful for the verification time to use such types for variables and literals. *Finally,* our CNF transformations can be linked with a verified CNF-SAT solver to obtain an end-to-end verified solver for the general SAT problem.

References

1. Cezar-Constantin Andrici and Ştefan Ciobâcă. Verifying the DPLL algorithm in Dafny. In Mircea Marin and Adrian Craciun, editors, *Proceedings Third Symposium on Working Formal Methods, FROM 2019, Timişoara, Romania, 3-5 September 2019*, volume 303 of *EPTCS*, pages 3–15, 2019.
2. Cezar-Constantin Andrici and Ştefan Ciobâcă. Who verifies the verifiers? A computer-checked implementation of the DPLL algorithm in Dafny. *CoRR*, abs/2007.10842, 2020.
3. Pedro Barroso, Mário Pereira, and António Ravara. Animated logic: Correct functional conversion to conjunctive normal form. In *PAAR 2020/SC-Square 2020*, volume 2752 of *CEUR Workshop Proceedings*, pages 1–20. CEUR-WS.org, 2020.
4. Ulrich Berger, Andrew Lawrence, Fredrik Nordvall Forsberg, and Monika Seisenberger. Extracting verified decision procedures: DPLL and resolution. *Log. Methods Comput. Sci.*, 11(1), 2015.
5. Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *J. Autom. Reasoning*, 61(1-4):333–365, 2018.
6. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. `https://hal.inria.fr/hal-00790310`.
7. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.
8. Thierry Boy de la Tour. An optimality result for clause form translation. *J. Symb. Comput.*, 14(4):283–302, 1992.
9. Mathias Fleury. Optimizing a verified SAT solver. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, pages 148–165, 2019.
10. Richard L. Ford and K. Rustan M. Leino. *Dafny Reference Manual.* 08 2017.
11. Lennard Gäher and Fabian Kunze. Mechanising Complexity Theory: The Cook-Levin Theorem in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
12. John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.
13. Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated

full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 165–181, 2014.

14. Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. SPARK 2014 and gnatprove - A competition report from builders of an industrial-strength verifying compiler. *Int. J. Softw. Tools Technol. Transf.*, 17(6):695–707, 2015.
15. K. Rustan M. Leino. Developing verified programs with Dafny. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1488–1490, 2013.
16. K. Rustan M. Leino. Accessible software verification with dafny. *IEEE Softw.*, 34(6):94–97, 2017.
17. Stephane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq.* Theses, Université Paris Sud - Paris XI, January 2011.
18. Filip Marić. Formalization and implementation of modern SAT solvers. *J. Autom. Reasoning*, 43(1):81–119, 2009.
19. Filip Marić and Predrag Janičić. Formalization of abstract state transition systems for SAT. *Log. Methods Comput. Sci.*, 7(3), 2011.
20. Julius Michaelis and Tobias Nipkow. Formalized proof systems for propositional logic. In *TYPES 2017*, volume 104 of *LIPIcs*, pages 5:1–5:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
21. Julius Michaelis and Tobias Nipkow. Propositional proof systems. *Archive of Formal Proofs*, June 2017. `https://isa-afp.org/entries/Propositional_Proof_Systems.html`, Formal proof development.
22. Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. versat: A verified modern SAT solver. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 363–378, 2012.
23. Natarajan Shankar and Marc Vaucher. The mechanical verification of a DPLL-based satisfiability solver. *Electr. Notes Theor. Comput. Sci.*, 269:3–17, 2011.
24. G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

A Short Overview of Dafny

The Dafny programming language supports auto-active verification, a technique known since the 70s, but which has only become practical in the past decade, thanks to advances in both the usability of verification tools and computer processing power. Here is an example of a simple typical auto-actively verified Dafny method implementing binary search:

```
method binarySearch(a: array<int>, key : int) returns (r : int)
   requires ∀ j, k • 0 ≤ j < k < a.Length ⟹ a[j] ≤ a[k];
   ensures r ≥ 0 ⟹ 0 ≤ r < a.Length ∧ a[r] = key;
   ensures r < 0 ⟹ ∀ k • 0 ≤ k < a.Length - 1 ⟹ a[k] ≠ key;
{
   var left : int := 0;
   var right : int := a.Length - 1;
   while (left ≤ right)
      invariant 0 ≤ left ≤ a.Length;
      invariant -1 ≤ right < a.Length;
      invariant ∀ k • 0 ≤ k < left ⟹ a[k] < key;
      invariant ∀ k • right < k < a.Length ⟹ a[k] > key;
      decreases right - left;
   {
      var mid : int := (left + right) / 2;
      if (key < a[mid]) {
         right := mid - 1;
      } else if (key > a[mid]) {
         left := mid + 1;
      } else {
         return mid;
      }
   }
   return -1;
}
```

Auto-active verification is a mix of *auto*matic verification and inter*active* verification. It means that the program is annotated by the programmer with specifications (such as preconditions, introduced by `requires`, and postconditions, introduced by `ensures`), which are automatically checked by the system to hold. The system verifies that the program implements the specification. If the postcondition cannot be proven to hold whenever the precondition holds, then the Dafny compiler fails with an error message. For example, the code above compiles (and verifies) without any issues; however, if we had made any mistake, like initializing `right` by `a.Length` instead of `a.Length - 1` or reversing the comparison operators `<` and `>` inside the body of the while loop, then compilation would fail as the postcondition would not be provable.
For complicated post-conditions, Dafny cannot establish their validity automatically, and therefore additional help is required from the part of the programmer (the interactive part) in the form of invariants (in the example above, four invariants are given for the while loop), variants (introduced by the `decreases` keyword, used to establish termination), lemmas, or other helper annotations.
Auto-active verification is featured in frameworks such as Why3 [6], Dafny [16] or even programming languages such as Ada [14] or C [7]. It has been used successfully to develop trusted code for small and even average sized projects [13]. The main advantage of using auto-active verification in software development is that we obtain a high degree of confidence in the correctness of software projects that were developed in this style.

B The Dafny Development

The attached Dafny development consists of 8 source files:
• `utils.dfy` contains generally useful definitions and lemmas, such as the definition of exponentiation (`pow`);
• `formula.dfy` contains the definition of the `FormulaT` data type and related functions and lemmas such as `validFormulaT`, `truthValue`, `maxVar`;
• `cnf.dfy` contains the verified implementation of the textbook algorithm for finding the CNF (with functions/methods such as `convertToCNF` or `applyRule`);
• `cnfformula.dfy` contains various items concerning the representation of CNF formulae as elements of type `seq<seq<int> >`, such as the predicates `validLiteral`, `validCnfFormula`, `truthValueCnfFormula`;

• `tseitin.dfy` contains the entry point (`tseitin`) to Tseitin's transformation, together with the main implementatino `tseitinCnf`;
• it relies on `tseitinproofs.dfy`, which contains lemmas that prove the invariant of `tseitinCnf` for all cases;
• both of the modules above rely on `tseitincore.dfy`, which contains definitions useful in both the algorithm and its proof, such as the set of clauses `orClauses` to be added for disjunctions;
• `main.dfy` exercises the CNF transformation in a `Main` method and can be used to obtain an executable;
• `Makefile` to be used in the usual Unix-like manner.
To compile (and verify) the development, it is sufficient to run:
• `dafny /verifySeparately *.dfy`.
We have verified the source code with Dafny version 3.0.0.20820, but some earlier versions should work as well.
The Dafny development is available at

`https://github.com/iordacheviorel/cnf-dafny`.