A Formalised Polynomial-Time Reduction From *3SAT* to *Clique*

Lennard Gäher
Saarland University, Germany
s8legaeh@stud.uni-saarland.de

Abstract

We present a formalisation of the well-known problems **SAT** and **Clique** from computational complexity theory. From there, a polynomial-time reduction from **3SAT**, a variant of **SAT** where every clause has exactly three literals, is developed and verified. All the results are constructively formalised in the proof assistant Coq, including the polynomial running time bounds. The machine model we use is the weak call-by-value lambda calculus.

1 Introduction

Since Karp presented his 21 *NP-complete* problems in 1972, the concept of NP-completeness has proved very fruitful in computational complexity theory. Many of his original reductions are nowadays taught in undergraduate courses on theoretical computer science. While the ideas these reductions rely on are very simple conceptually, the proofs of correctness are done with a lot of handwaving, relying on the reader's intuition. The reasons for this are diverse: Turing machines, the standard computatinal model in the theory of NP-completeness, are very low-level and non-compositional, making formal arguments about their behaviour quite painful. Moreover, formal running time analyses are often deemed unnecessary, since the involved machines only need to run in polynomial time – which can usually be readily believed.

Nevertheless, a formal treatment of the theory of NP-completeness is desirable: Verifying these proofs that have been taught for decades might yield new insights into invariants which are tacitly assumed.

In this memo, we provide a formalisation of the **SAT** and **Clique** problems, prove that they are in *NP* and provide a formalisation of a well-known reduction from **3SAT** to **Clique**. We use a weak call-by-value lambda calculus called L introduced by Forster and Smolka in [3] as the model of computation. L has been proved to be a reasonable computational model by Forster et al. in [2], in the sense that L and Turing machines can simulate each other with a polynomial overhead in time and a constant overhead in space. While the lambda-calculus is higher-level than Turing machines, directly programming in L is still unpleasant. Forster and Kunze have provided a framework for the certifying extraction of Coq terms to L terms, which also allows the user to instantiate time bounds during the extraction process [1]. Thus we can formalise problems and reductions in Coq and later extract these definitions to L, proving running time bounds in the process.

2 SAT

When formalising computational problems, one has to strike a balance regarding abstraction and technical sophistication: While reasoning about correctness in Coq can be done in a much more elegant way when using abstract formalisations and inductive predicates instead of computational functions, members of $\mathbb{P}$, the universe of propositions, cannot be extracted to L. Running time analyses get easier, too, when employing simple formalisations.

We therefore define most properties using inductive predicates and accompany them with Boolean deciders if necessary, while all datatypes used for the problem definitions are kept

as simple as possible.

2.1 Conjunctive Normal Forms

Following this line of thought, we define a simple list-based representation of conjunctive normal forms (CNFs):

$$x : \mathsf{var} := \mathsf{N} \qquad L : \mathsf{literal} := \mathsf{B} \times \mathsf{var} \qquad C : \mathsf{clause} := \mathcal{L}\ \mathsf{literal} \qquad N : \mathsf{cnf} := \mathcal{L}\ \mathsf{clause}$$

A literal is represented by a pair of a Boolean and a natural number, denoting its sign and its variable. Assignments are lists of Booleans: $a : \mathsf{assgn} := \mathcal{L}\ \mathsf{B}$. The list $[\mathsf{T}, \mathsf{F}, \mathsf{T}]$, for instance, denotes the assignment $\{x_0 \mapsto 1, x_1 \mapsto 0, x_2 \mapsto 1\}$.

The evaluation of CNFs is defined via Boolean functions recursively defined on the structure of the lists:

$$([]).0 := \emptyset \qquad (x :: xs).0 := {}^\circ x \qquad ([]).(Sn) := \emptyset \qquad (x :: xs).(Sn) := xs.n$$

$$\begin{aligned}
&\mathsf{foldrO} : (B \to A \to \mathcal{O}(A)) \to \mathcal{O}(A) \to \mathcal{L}(B) \to \mathcal{O}(A)\\
&\mathsf{foldrO}\ f\ acc\ [] := acc\\
&\mathsf{foldrO}\ f\ acc\ (x :: xs) := \mathbf{match}\ \mathsf{foldrO}\ f\ acc\ xs\ [\ {}^\circ acc \Rightarrow (f\ x\ acc) \quad | \quad \emptyset \Rightarrow \emptyset\ ]
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\ a\ L\ &:= \mathbf{let}\ (s, n) := L\ \mathbf{in}\ \mathbf{match}\ a.n[\ {}^\circ value \Rightarrow {}^\circ(s \stackrel{?}{=} value) \quad | \quad \emptyset \Rightarrow \emptyset\ ]\\
\mathcal{E}\ a\ C\ &:= \mathsf{foldrO}\ (\lambda\ L\ acc.\ \mathbf{match}\ \mathcal{E}\ a\ L\ [\ {}^\circ v \Rightarrow {}^\circ(acc\ ||\ v) \quad | \quad \emptyset \Rightarrow \emptyset\ ])\ ({}^\circ\mathsf{F})\ C\\
\mathcal{E}\ a\ N\ &:= \mathsf{foldrO}\ (\lambda\ C\ acc.\ \mathbf{match}\ \mathcal{E}\ a\ C\ [\ {}^\circ v \Rightarrow {}^\circ(acc\ \&\&\ v) \quad | \quad \emptyset \Rightarrow \emptyset\ ])\ ({}^\circ\mathsf{T})\ N
\end{aligned}$$

The function $xs.n$ yields the n-th element of xs, wrapped in the option type $\mathcal{O}$, or $\emptyset$ if the subscript is invalid, and is used to lookup values of assignments. foldrO is a variant of the known fold-right function, but the aggregation function returns an element of $\mathcal{O}(A)$. The folding only continues as long as the aggregation result is not $\emptyset$. Using foldrO, the definition of the evaluation function $\mathcal{E}$ is straightforward. The presence of option values is due to our choice that an assignment only has finite support, which we find more expressive than using default values. We use $\mathcal{E}$ to refer to evaluation of both literals, clauses and CNFs, but from the context and the metavariables it will always be clear which of the functions we mean.

Next we introduce the notion of boundedness of CNFs: a CNF N is bounded by k, written $N \prec k$, if all variables occuring in N are strictly smaller than k. We need the same notion for clauses and again use the same notation to refer to boundedness of clauses and CNFs. This is formalised using inductive predicates:

$$\frac{}{[] \prec k} \qquad \frac{n < k \quad C \prec k}{((s, n) :: C) \prec k} \qquad \frac{C \prec k \qquad N \prec k}{C :: N \prec k}$$

▶ **Proposition 1.**
1. $k \leq k' \to C \prec k \to C \prec k'$
2. $k \leq k' \to N \prec k \to C \prec k'$

▶ **Proposition 2.** *If the CNF is bounded by the length $|a|$ of the assignment, evaluation will yield a result.*
1. $v < k \to |a| \geq k \to \exists\ r.\mathcal{E}\ a\ (s, v)\ = {}^\circ r$
2. $C \prec k \to |a| \geq k \to \exists\ r.\mathcal{E}\ a\ C\ = {}^\circ r$
3. $N \prec k \to |a| \geq k \to \exists\ r.\mathcal{E}\ a\ N\ = {}^\circ r$

Proof. 1. is proved using standard facts on $xs.n$. 2. and 3. are proved by induction on $C \prec k$ and $N \prec k$, respectively. In the inductive step of 2., we need 1., while we need 2. in the inductive step of 3. ◀

The converse statements of Proposition 2 also hold.

▶ **Proposition 3.** *If evaluation succeeds for an assignment a, the CNF is bounded by $|a|$.*
1. $(\exists\, r.\mathcal{E}\ a\ (s, v)\ = {}^\circ r) \to v < |a|$
2. $(\exists\, r.\mathcal{E}\ a\ C\ = {}^\circ r) \to C \prec |a|$
3. $(\exists\, r.\mathcal{E}\ a\ N\ = {}^\circ r) \to N \prec |a|$

Apart from the inductive predicate $\prec$, we will also need a computable notion of boundedness which can be extracted to L. We use functions maxClause and maxCnf that compute the maximal variable which is used in a clause or CNF, respectively:

$$\mathsf{maxClause}\ C := \mathsf{foldr}\ (\lambda\ (\_, n)\ acc.\ \max acc\ n)\ 0\ C$$
$$\mathsf{maxCnf}\ N := \mathsf{foldr}\ (\lambda\ C\ acc.\ \max acc\ (\mathsf{maxClause}\ C))\ 0\ N$$

▶ **Proposition 4.** *A CNF is bounded by the successor of the maximal variable.*
1. $C \prec (1 + \mathit{maxClause}\ C)$
2. $N \prec (1 + \mathit{maxCnf}\ N)$

Proof. 1. The statement is shown using induction on C. In the inductive step, the inductive definition of $\prec$ leaves us with two proof obligations. For the second obligation $C \prec (1 + \mathsf{maxClause}(L :: C))$ for some literal L, we need the monoticity given by Proposition 1.
2. Similar to 1.

◀

We proceed with results on the evaluation functions.

▶ **Lemma 5.** *One evaluation step can be characterised as follows:*
1. $\mathcal{E}\ a\ (L :: C)\ = {}^\circ b \leftrightarrow \exists\, b_1, b_2.\mathcal{E}\ a\ C\ = {}^\circ b_1 \wedge \mathcal{E}\ a\ L\ = {}^\circ b_2 \wedge b = b_1\ ||\ b_2$
2. $\mathcal{E}\ a\ (C :: N)\ = {}^\circ b \leftrightarrow \exists\, b_1, b_2.\mathcal{E}\ a\ N\ = {}^\circ b_1 \wedge \mathcal{E}\ a\ C\ = {}^\circ b_2 \wedge b = b_1\ \&\&\ b_2$

Proof. Using induction on C and N, respectively. A few case analyses are needed, but otherwise the proof is standard. ◀

We can now derive a helpful equivalent characterisation of satisfaction of clauses and CNFs:

▶ **Lemma 6.**
1. $\mathcal{E}\ a\ C\ = {}^\circ\mathsf{T} \leftrightarrow (\exists\, L \in C.\mathcal{E}\ a\ L\ = {}^\circ\mathsf{T}) \wedge C \prec |a|$
2. $\mathcal{E}\ a\ N\ = {}^\circ\mathsf{T} \leftrightarrow (\forall\, C \in N.\mathcal{E}\ a\ C\ = {}^\circ\mathsf{T}) \wedge N \prec |a|$

Proof. 1. By induction on C. The base case is trivial. In the direction from left to right in the inductive step, the statement $L :: C \prec |a|$ follows directly from Proposition 3. For the other part of the conjunction, we apply Lemma 5. For the direction from right to left, we need Proposition 2 and a few case analyses.
2. By induction on N. The proof is simpler than the one for 1. In the inductive step, we need Lemma 5.

◀

2.2 Satisfiability

We now define **SAT** as the following problem:

▶ **Definition 7.** *Given a CNF N, **SAT** is the problem of determining whether there exists a satisfying assignment a:*

$$\boldsymbol{SAT}\ N := \exists\ a.\mathcal{E}\ a\ N\ = {}^\circ\mathsf{T}$$

In the remainder of this section, we show that **SAT** is in NP. We use the well-known definition of NP using polynomial-time verifiers, since this saves us from introducing non-determinism to L. The verifier gets a problem instance and a certificate and has to check whether the certificate is a valid proof of the instance being an inhabitant of the problem at hand. There should be valid certificates exactly for the positive instances. We require that a verifier only accepts certificates having an encoding size polynomial in the encoding size of the instance[1].

Using the functions defined above, it is straightforward to define a verifier for **SAT**:

$$\mathsf{Ver}_{\mathbf{SAT}}\ N\ a := \mathcal{E}\ a\ C\ = {}^\circ\mathsf{T} \wedge |a| \leq 1 + \mathsf{maxCnf}\ N$$

Note that this is again a propositional definition. A Boolean decider, which can then be extracted to L, can be derived in a straightforward way, though. The second condition captures the requirement of polynomial certificates: Valid certificates are only as long as they need to be. Nevertheless, it isn't straightforward to see that the requirement is actually fulfilled: It only works because the Scott encoding of natural numbers which we use is unary in nature. Since the encoding size of Booleans is constant, the encoding size of a minimal satisfying assignment is thus linear in the encoding size of the maximal variable used, which is part of the encoding of the CNF.

In order to derive that **SAT** is in NP, we need to show that:

1. $\mathbf{SAT}\ N \leftrightarrow \exists\ a.\mathsf{Ver}_{\mathbf{SAT}}\ N\ a$,
2. the Boolean version of $\mathsf{Ver}_{\mathbf{SAT}}$ runs in polynomial time in the encoding size of its two inputs,
3. $\mathsf{Ver}_{\mathbf{SAT}}$ only accepts certificates whose encoding size is polynomial in the encoding size of N.

Since 2. and 3. are quite technical to prove, we only go into step 1. in detail. Essentially, we need to show that if there exists a certificate, i.e. a satisfying assignment, then there also exists a short one. This short assignment can be obtained by only taking a prefix of the assignment. The standard $\mathsf{take} : \mathcal{L}\ A \to \mathsf{N} \to \mathcal{L}\ A$ function is used.

▶ **Lemma 8.**

1. $n < k \to \mathcal{E}\ a\ (s,n)\ = {}^\circ v \to \mathcal{E}\ (\mathsf{take}\ a\ k)\ (s,n)\ = {}^\circ v$
2. $C \prec k \to \mathcal{E}\ a\ C\ = {}^\circ v \to \mathcal{E}\ (\mathsf{take}\ a\ k)\ C\ = {}^\circ v$
3. $N \prec k \to \mathcal{E}\ a\ N\ = {}^\circ v \to \mathcal{E}\ (\mathsf{take}\ a\ k)\ N\ = {}^\circ v$

Proof. 1. follows from $n < k \to l.n = {}^\circ v \to (\mathsf{take}\ l\ k).n = {}^\circ v$, which is proved using standard facts on take and $l.n$. 2. and 3. follow by induction on C and N, respectively. ◀

[1] An alternative definition not posing this restriction, but instead requiring that there exists a certificate of polynomial size exactly for the positive instances would also work, but make proofs more complicated.

2.3 k-Satisfiability

We introduce the notion of k-CNFs using an inductive predicate.

$$\frac{k > 0}{\mathsf{kCNF}\ k\ []} \qquad \frac{|C| = k \quad \mathsf{kCNF}\ k\ N}{\mathsf{kCNF}\ k\ (C :: N)}$$

Thus, the representation of k-CNFs does not differ from the representation of unconstrained CNFs.

▶ **Proposition 9.** *kCNF* $k\ N \leftrightarrow k > 0 \wedge \forall C \in N.k = |C|$

The corresponding problem is k-**SAT**:

▶ **Definition 10.** *For every fixed k, k-**SAT** is the problem of determining whether there exists a satisfying assignment a for a k-CNF N:*

$$k\text{–}\boldsymbol{SAT}\ N := \mathit{kCNF}\ k\ N \wedge \exists\ a.\mathcal{E}\ a\ N\ \ = {}^{\circ}\mathsf{T}$$

A reduction of k-**SAT** to **SAT** can be easily obtained and formalised.

3 Clique

In this section, we formalise the **Clique** problem.

3.1 Graphs

We start with a formalisation of undirected graphs which can be extracted to L. Again, we strive to keep the representation as simple as possible in order to admit a compact running time analysis, trading off expressivity. Thus we use a list-based representation.

$$n, u, v : \mathsf{node} := \mathsf{N} \qquad e : \mathsf{edge} := \mathsf{node} \times \mathsf{node} \qquad g : \mathsf{graph} := \mathsf{N} \times \mathcal{L}\ \ \mathsf{edge}$$

A graph is pair of a natural number giving the number of nodes and a list of edges. This definition is very weak and doesn't give any semantic guarantees. We correct for this lack by giving an inductive predicate defining well-formed graphs:

$$\frac{}{\mathsf{gwf}\ (n, [])} \qquad \frac{u < n \quad v < n \quad \mathsf{gwf}\ (n, e)}{\mathsf{gwf}\ (n, (u, v) :: e)}$$

We give Boolean deciders for node and edge containment:

$$\mathsf{nodeIn}\ g\ n := \mathbf{let}\ (max, \_) := g\ \mathbf{in}\ (1 + n) \overset{?}{\leq} max$$
$$\mathsf{edgeIn}\ g\ u\ v := \mathbf{let}\ (\_, e) := g\ \mathbf{in}\ (u, v) \overset{?}{\in} e\ ||\ (v, u) \overset{?}{\in} e$$

While our definition of the type edge works with pairs, we are giving it the semantics of an undirected graph.

3.2 Clique

A k-clique is a duplicate-free list of k nodes such that all pairwise-distinct nodes are connected.

$$\frac{}{\mathsf{isClique}\ g\ []\ 0} \qquad \frac{n \notin cl \quad \mathsf{nodeIn}\ g\ n = \mathsf{T} \quad (\forall n' \in cl.\mathsf{edgeIn}\ g\ n\ n' = \mathsf{T}) \quad \mathsf{isClique}\ g\ cl\ k}{\mathsf{isClique}\ g\ (n :: cl)\ (1+k)}$$

▶ **Lemma 11** (Explicit characterisation)**.**

$$\mathsf{isClique}\ g\ cl\ k \leftrightarrow |cl| = k \wedge \mathsf{dupfree}\ cl \wedge (\forall n \in cl.\mathsf{nodeIn}\ g\ n = \mathsf{T}) \\ \wedge (\forall u, v \in cl.u \neq v \rightarrow \mathsf{edgeIn}\ g\ u\ v = \mathsf{T})$$

Proof. The direction from left to right follows by an induction on the derivation of isClique. The other direction follows by an induction on cl. ◀

We can now define the **Clique** problem.

▶ **Definition 12.** *Given a graph g and a natural number $k > 0$, **Clique** is the problem of determining whether there is a set of nodes cl such that cl is a k-clique in g:*

$$\mathbf{Clique}(g, k) := \exists\ cl.\mathsf{isClique}\ g\ cl\ k$$

A verifier can be derived mechanically, using the list of nodes as the certificate:

$$\mathsf{Ver}_{\mathbf{Clique}}((g, k), cl) := \mathsf{isClique}\ g\ cl\ k$$

In contrast to the **SAT** verifier, we need not explcitly require that the certificate is short enough, since already isClique places a bound. At this point it is critical that the representation of graphs explicitly contains a bound on the indices of nodes.

It is straightforward to obtain a Boolean decider for the isClique predicate and thus a computable verifier.

4 Reducing 3SAT to Clique

In this section, we finally reduce **3SAT** to **Clique**. The reduction is well-known; nevertheless, we give a short account of the intuition behind it.

Given a **3SAT** instance N, we have to construct a graph g and a number k such that g has a k-clique if, and only if, N is satisfiable. First observe that an assignment satisfies N iff for every clause $C \in N$, there is at least one literal $L \in C$ such that $\mathcal{E}\ a\ L\ = °\mathsf{T}$ (this is a direct consequence of the two statements in Lemma 6). We therefore construct a graph with $3 \cdot |N|$ nodes, one for each literal: $n_0^0, n_1^0, n_2^0, n_0^1, \ldots, n_2^{|N|}$. Two nodes n_j^i, n_l^k are connected via an edge iff $i \neq k$, i.e. the corresponding literals belong to distinct clauses, and the literals are not conflicting, meaning that there is an assignment that satisfies both literals simultaneously.

The graph now has a $|N|$ clique iff N is satisfiable. It is easy to believe (although not formally satisfactory) that this construction does indeed satsisfy the desired equivalence.

▶ **Definition 13** (Conflicting literals)**.**

$$\mathsf{conflict}\ (s_1, n_1)\ (s_2, n_2) := s_1 \neq s_2 \wedge n_1 = n_2$$

▶ **Proposition 14.**

$$n_1 < |a| \rightarrow n_2 < |a| \rightarrow (\mathsf{conflict}\ (s_1, n_1)\ (s_2, n_2) \leftrightarrow \mathcal{E}\ a\ (s_1, n_1)\ \neq \mathcal{E}\ a\ (s_2, n_2)\ )$$

In order to reason about the reduction, we have to somehow connect a CNF with the corresponding graph. Since there is a one-to-one correspondence between literals of the CNF and nodes of the graph, we require a bijection between nodes and literals. However, there can be several literals in one CNF that are syntactically identical but are at different positions in the list representation, meaning that we cannot simply use elements of literal. Instead, we use pairs of indices describing the index of the clause and the index of the literal inside this clause.

▶ **Definition 15** (Bijections of sets). *Let two sets be given by* $p : X \to \mathbf{P}$ *and* $q : Y \to \mathbf{P}$*. Let* $f : X \to Y$ *and* $g : Y \to X$*.*

$$\mathsf{inverseOn}\ p\ q\ f\ g := (\forall x.p\ x \to q(f\ x) \land x = g(f\ x)) \land (\forall y.q\ y \to p(f\ y) \land y = f(g\ y))$$

▶ **Definition 16** (Labellings of graphs). *A labelling assigns to each node of a graph a corresponding literal, and vice versa.*

$$\mathsf{labG} := \mathsf{N} \to \mathsf{N} \times \mathsf{N} \qquad \mathsf{labG}^{-1} := \mathsf{N} \times \mathsf{N} \to \mathsf{N}$$

$$\mathsf{isLabelling}\ N\ (f : \mathsf{labG})\ (f^{-1} : \mathsf{labG}^{-1}) := \mathsf{inverseOn}\ f\ f^{-1}\ (\lambda\ v.v < 3 \cdot |N|)\ (\lambda\ (c,l).c < |N| \land l < 3)$$

From the literal positions, we can restore the syntax of the literal:

$$\mathsf{enumLiteral}\ N\ (cl,l) := \mathbf{match}\ N.cl\ [\ {}^\circ C \Rightarrow C.l \quad | \quad \emptyset \Rightarrow \emptyset\ ]$$

$$L\ \mathsf{inCnf}\ N := \exists\ C.C \in N \land L \in C$$

▶ **Proposition 17** (Correctness of enumLiteral).
1. $\mathsf{kCNF}\ k\ N \to cl < |N| \to l < k \to \exists\ L.\mathsf{enumLiteral}\ N\ (cl,l) = {}^\circ L \land L\ \mathsf{inCnf}\ N$
2. $\mathsf{kCNF}\ k\ N \to (\exists\ L.\mathsf{enumLiteral}\ N\ (cl,l) = {}^\circ L) \to cl < |N| \land l < k$

Proof. Both statements are proved using induction on N and standard facts on $xs.n$. For the first statement, we need to quantify over n. ◀

Using literal positions, we can formalise what it means that two literals are part of another clause:

$$\mathsf{diffClause}\ (cl_1,l_1)\ (cl_2,l_2) := cl_1 \neq cl_2$$

4.1 A Reduction Relation

We first define a relation $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$ which connects positive **3SAT** instances to corresponding graphs. This relation abstracts away from specific syntactic properties of the graph, such as the order of the nodes or the order in which edges are stored. We show that two **3SAT** and **Clique** instances N and (g,k) which are related by this relation are equivalent, in the sense that one instance is an inhabitant of the problem iff the other instance is an inhabitant of the other problem: $N \sim^{\mathbf{3SAT}}_{\mathbf{Clique}} (g,k) \to (\mathbf{SAT}\ N \leftrightarrow \mathbf{Clique}\ (g,k))$.

▶ **Definition 18.** *Let* $N : \mathit{cnf}$ *and* $((n,e),k) : \mathit{graph} \times \mathsf{N}$*.* N *and* $((n,e),k)$ *are related, written* $N \sim^{\mathit{3SAT}}_{\mathit{Clique}} ((n,e),k)$*, iff*
- N *is a 3-CNF:* $\mathsf{kCNF}\ 3\ N$
- $n = 3 \cdot |N| \land k = |N|$

- *there is a labelling* (f, f^{-1}), $\mathsf{isLabelling}\ N\ f\ f^{-1}$, *such that for all* $u, v < n$, $\mathsf{edgeIn}\ (n, e)\ u\ v = \mathsf{T}$ *is equivalent to*

$$\begin{aligned}&\mathsf{diffClause}\ (f\ u)\ (f\ v)\\ \wedge\forall L_1, L_2.&\mathsf{enumLiteral}\ N\ (f\ u) = {}^\circ L_1 \rightarrow \mathsf{enumLiteral}\ N\ (f\ v) = {}^\circ L_2 \rightarrow \neg(\mathsf{conflict}\ L_1\ L_2)\end{aligned}$$

This relation exactly captures the construction described above, with the minor technical nuisance of having to convert between literal positions and syntactical literals.

We show the above equivalence for related instances constructively, by explicitly constructing a clique for a satisfying assignment, and a satisfying assignment for a clique.

4.2 Constructing a Clique From a Satisfying Assignment

We start with the construction of cliques. If we have an assignment a satisfying the CNF N, the corresponding clique in a graph g related by $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$ consists of one node per clause which is satisfied by the assignment. The construction proceeds by recursively iterating over the CNF and checking which literal is satisfied. We have to keep track of the clause and literal indices while doing so.

$$\begin{aligned}&\mathsf{constrCliCla'}\ a\ cl\ []\ l := \emptyset\\ &\mathsf{constrCliCla'}\ a\ cl\ (L :: C)\ l := \mathbf{match}\ \mathcal{E}\ a\ L\\ &\qquad\qquad [\ {}^\circ\mathsf{T} \Rightarrow {}^\circ(cl, l)\quad |\quad \_ \Rightarrow \mathsf{constrCliCla'}\ a\ cl\ C\ (1 + l)\ ]\\ &\mathsf{constrCliCla}\ a\ cl\ C := \mathsf{constrCliCla'}\ a\ cl\ C\ 0\end{aligned}$$

$$\begin{aligned}&\mathsf{constrCliCnf'}\ a\ []\ cl := []\\ &\mathsf{constrCliCnf'}\ a\ (C :: N)\ cl := \mathbf{match}\ \mathsf{constrCliCla}\ a\ C\ cl\\ &\qquad\qquad [\ {}^\circ p \Rightarrow p :: \mathsf{constrCliCnf'}\ a\ N\ (1 + cl)\quad |\quad \emptyset \rightarrow []\ ]\\ &\mathsf{constrCliCnf}\ a\ N := \mathsf{constrCliCnf'}\ a\ N\ 0\end{aligned}$$

The function constrCliCnf calculates literal positions. If we map those positions to a graph (using a labelling) related to the CNF via $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$, we get a clique.

▶ **Lemma 19.** *If* a *satisifies* N, constrCliCnf will indeed yield a literal position for every clause.

1. $\mathcal{E}\ a\ N\ = {}^\circ\mathsf{T} \rightarrow C \in N \rightarrow \exists\ l.\mathsf{constrCliCla}\ a\ cl\ C = {}^\circ(cl, l)$
2. $\mathcal{E}\ a\ N\ = {}^\circ\mathsf{T} \rightarrow cl < |N| \rightarrow \exists\ l.(cl, l) \in \mathsf{constrCliCnf}\ a\ N$

Proof. 1. We generalise over the constant 0 in the definition and then do an induction over C. In the inductive step, we need Lemma 5, statement 1.

2. By induction on N, generalising over the constant 0. In the inductive step, we use the first statement and Lemma 5, statement 2.

◀

▶ **Lemma 20.** *Every two members of the constructed position list describe literals of different clauses.*

$$(\mathsf{constrCliCnf}\ a\ N).i = {}^\circ pos \rightarrow (\mathsf{constrCliCla}\ a\ N).j = {}^\circ pos' \rightarrow i \neq j \rightarrow \mathsf{diffClause}\ pos\ pos'$$

Proof. By generalising over the constant 0 and induction over N. For the inductive step, we need several auxiliary lemmas giving bounds on the indices which are part of the result list. ◀

While the above result isn't very helpful on its own, we get two important corollaries from it.

▶ **Corollary 21.**

$pos \in \mathsf{constrCliCnf}\ a\ N \to pos' \in \mathsf{constrCliCla}\ a\ N \to pos \neq pos' \to \mathsf{diffClause}\ pos\ pos'$

▶ **Corollary 22.** $\mathsf{dupfree}\ (\mathsf{constrCliCnf}\ a\ N)$

▶ **Lemma 23.** *All literals determined by* constrCliCnf are satisfied by the assignment.

$pos \in \mathsf{constrCliCnf}\ a\ N \to \exists\ L.\mathsf{enumLiteral}\ N\ pos = {}^\circ L \wedge \mathcal{E}\ a\ L\ = {}^\circ\mathsf{T}$

▶ **Proposition 24.** *Every two literals determined by* constrCliCnf are non-conflicting.

$pos, pos' \in \mathsf{constrCliCnf}\ a\ N \to pos \neq pos'$
$\to \exists\ L, L'.\mathsf{enumLiteral}\ N\ pos = {}^\circ L \wedge \mathsf{enumLiteral}\ N\ pos' = {}^\circ L' \to \neg(\mathsf{conflict}\ L\ L')$

Proof. By Lemma 23 and Proposition 14. ◀

4.3 Constructing a Satisfying Assignment From a Clique

In this subsection, we deal with the other direction of the equivalence. Starting with a clique in a graph g connected to a CNF N via $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$, we have to show the existence of a satisfying assignment. Again, we do this constructively by computing such an assignment. Due to the construction of $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$, all members of a clique need to correspond to literals of different clauses, and all of these literals are non-conflicting. Thus we obtain a satisfying assignment by setting all these literals to T and assigning arbitrary values to all unaffected variables.

From a technical perspective, this direction is more involved: Due to the inherent asymmetry of $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$, we have to directly reason about properties of the graph which are asserted by $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$. We therefore proceed in four steps, changing the representation in each until we end up with a satisfying assignment:

1. We start with a graph g, CNF N and a $|N|$-clique cl of g such that $N \sim^{\mathbf{3SAT}}_{\mathbf{Clique}} (g, |N|)$.
2. This is translated to a list of literal positions ($\mathcal{L}\ \ (\mathsf{N} \times \mathsf{N})$), giving the positions of the literals corresponding to the nodes in cl. The literals at these positions are non-conflicting and there is exactly one for every clause of N.
3. We proceed by mapping this list to a list of syntactic literals. These are non-conflicting and if all of them are satisfied by an assignment, this assignment also satisfies the CNF.
4. The list in 3. can be interpreted as a partial assignment. If this is expanded to a complete assignment, assigning arbitrary values to all other variables, we get a satisfying assignment.

4.3.1 Mapping to Literal Positions

Throughout this subsection, we assume a CNF N, a graph g and a $|N|$-clique cl of g, such that $N \sim^{\mathbf{3SAT}}_{\mathbf{Clique}} (g, |N|)$. Let (f, f^{-1}) be the labelling from $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$. We simply apply the labelling function to all elements of the clique:

$\mathsf{toPos}\ cl := \mathsf{map}\ f\ cl$

From the reduction relation, we get a number of straightforward properties:

▶ **Lemma 25.** *Let N be a CNF, g a graph such that $N \sim^{\textbf{3SAT}}_{\textbf{Clique}} (g, |N|)$.*

1. $(a, b) \in \mathsf{toPos}\ cl \to a < |N| \wedge b < 3$
2. $pos \in \mathsf{toPos}\ cl \to \exists\ L.\mathsf{enumLiteral}\ N\ pos = {}^\circ L$
3. $pos, pos' \in \mathsf{toPos}\ cl \to pos \neq pos'$
   $\to \exists\ L, L'.\mathsf{enumLiteral}\ N\ pos = {}^\circ L \wedge \mathsf{enumLiteral}\ N\ pos' = {}^\circ L' \to \neg(\mathsf{conflict}\ L\ L')$

▶ **Lemma 26.** *For every clause of N, there is a corresponding literal position in the result of* toPos*: $k < |N| \to \exists\ l.(k, l) \in \mathsf{toPos}\ cl$*

Proof. Essentially, the statement is an instance of the pigeon-hole principle: There are $|N|$ nodes in the clique cl and $|N|$ clauses. Since no two distinct nodes of cl belong to the same clause, there must be one for every clause. Proving this formally requires somewhat more effort.

First of all, we can constructively do a proof by contradiction, i.e. assume that $\neg(\exists\ l.(k, l) \in \mathsf{toPos}\ cl)$, since the possible range of l is finite by Lemma 25, part 1.

It now suffices to show that there is a clause index that occurs twice in the output of toPos, $\mathsf{rep}\ (\mathsf{map}\ \mathsf{fst}\ (\mathsf{toPos}\ cl))$. For, if this is the case, there are indices i_1, i_2, a clause index k' and literal indices l_1, l_2 such that

$$i_1 \neq i_2 \wedge (\mathsf{toPos}\ cl).i_1 = {}^\circ(k', l_1) \wedge (\mathsf{toPos}\ cl).i_2 = {}^\circ(k', l_2).$$

Now, if $l_1 = l_2$, then there was a duplicate in the clique cl, since the labelling f is injective; but this is a contradiction to the definition of a clique. Otherwise, $l_1 \neq l_2$ and we get a contradiction by Lemma 25, part 3.

Thus we show $\mathsf{rep}\ (\mathsf{map}\ \mathsf{fst}\ (\mathsf{toPos}\ cl))$. An instance of the pigeon-hole principle on lists reads as follows: $\forall l_1, l_2.l_1 \subseteq l_2 \to |l_2| < |l_2| \to \mathsf{rep}\ l_1$. Since by assumption and Lemma 25, part 1, $\mathsf{map}\ \mathsf{fst}\ (\mathsf{toPos}\ cl) \subseteq [0, \ldots, |N| - 1] \setminus [k]$, but the former list has $|N|$ elements, this closes the proof. ◀

4.3.2 Mapping to Literals

Throughout this subsection, we assume that the list *posl* is the output of toPos of the previous subsection, thus satisfying the properties proved for it.

We define two functions which use enumLiteral to obtain the syntactic representations of literals referenced by literal positions.

$$\mathsf{toLit'}\ posl := \mathsf{map}\ (\mathsf{enumLiteral}\ N)\ posl$$
$$\mathsf{toLit}\ posl := \mathsf{foldr}\ (\lambda\ l\ acc.\mathbf{match}\ l\ [\ {}^\circ a \Rightarrow a :: acc \quad |\quad \_ \Rightarrow acc\ ])\ []\ (\mathsf{toLit'}\ posl)$$

The function toLit strips away the option wrappers from the output of toLit', a procedure which is justified by the following lemmas:

▶ **Lemma 27.**

1. $L' \in \mathsf{toLit'}\ posl \to \exists\ L.L' = {}^\circ L$
2. $p \in posl \to \mathsf{enumLiteral}\ c\ a \neq \emptyset$
3. $L \in \mathsf{toLit}\ posl \to L\ \mathsf{inCnf}\ N$
4. $L \in \mathsf{toLit}\ posl \to \exists\ pos.pos \in posl \wedge \mathsf{enumLiteral}\ N\ pos = {}^\circ L$
5. ${}^\circ L \in \mathsf{toLit'}\ posl \to L \in \mathsf{toLit}\ posl$
6. $L_1, L_2 \in \mathsf{toLit}\ posl \to \neg(\mathsf{conflict}\ L_1\ L_2)$

Proof. All of the statements follow straightforwardly from the definitions. ◀

▶ **Lemma 28.** *Satisfying the literals given by* toLit is enough to obtain a satisfying assignment.
$(\forall L \in \mathsf{toLit}\ posl.\ \mathcal{E}\ a\ L\ = °\mathsf{T}) \to N \prec |a| \to \mathcal{E}\ a\ N\ = °\mathsf{T}$

Proof. We use the characterisation of evaluation given by Lemma 6. The statement then needs to be strenghtened in order to obtain additional information on the index of the clauses:

$$\forall n < |N|.\exists\ \mathrm{C}.\ N.n = °C \wedge \mathcal{E}\ a\ C\ = °\mathsf{T}$$

Using Lemma 6, the rest of the proof is easy. ◀

4.3.3 Expanding to a Full Assignment

The only thing that is left to do is to expand a partial assignment $A : \mathcal{L}\ \mathsf{literal}$ given by a list of non-conflicting literals which should be satisfied to a full assignment a. For this, we define a recursive function which takes a variable index *largestVar* up to which the constructed assignment should be valid and a partial assignment p. We again assume that p satisfies the properties proved for the output of toPos in the previous subsection.

$$\begin{aligned}
&\mathsf{lookup}\ m\ [] := \emptyset \\
&\mathsf{lookup}\ m\ ((s,n) :: p) := \mathbf{if}\ n \stackrel{?}{=} m\ \mathbf{then}\ °s\ \mathbf{else}\ \mathsf{lookup}\ m\ p
\end{aligned}$$

$$\begin{aligned}
\mathsf{expand}\ \mathit{largestVar}\ p :=& (\mathbf{match}\ \mathit{largestVar}\ [\ 0 \Rightarrow []\ \ |\ \ S\ l \Rightarrow \mathsf{expand}\ l\ p\ ]) \\
&+\!\!+ [\mathbf{match}\ \mathsf{lookup}\ \mathit{largestVar}\ p\ [\ °\,b \Rightarrow b\ \ |\ \ \emptyset \Rightarrow \mathsf{F}\ ]]
\end{aligned}$$

▶ **Proposition 29.** *The value returned by* lookup is the first value of the list l for which the associated variable matches the requested variable n.

$$\mathsf{lookup}\ n\ l = °s \leftrightarrow \exists\ i.\ l.i = °(s,n) \wedge \forall j.j < i \to \forall s'.\ l.j \neq °(s',n)$$

▶ **Lemma 30.**

1. $|\mathsf{expand}\ lV\ p| = 1 + lV$
2. $\mathsf{expand}\ (1 + lV)\ p = \mathsf{expand}\ lV\ p +\!\!+ [\mathbf{match}\ \mathsf{lookup}\ (1 + lV)\ p\ [\ °b \Rightarrow b\ \ |\ \ \emptyset \Rightarrow \mathsf{F}\ ]]$
3. $m \leq lV \to (\mathsf{expand}\ lV\ p).m = °(\mathbf{match}\ \mathsf{lookup}\ m\ p\ [\ °b \Rightarrow b\ \ |\ \ \emptyset \Rightarrow \mathsf{F}\ ])$
4. $m \neq n \to m \leq lV \to (\mathsf{expand}\ lV\ ((s,n) :: p)).m = (\mathsf{expand}\ lV\ p).m$
5. $m \leq lV \to (n = m \to (\mathsf{expand}\ lV\ ((s,n) :: p)).m = °s)$
   $\wedge (n \neq m \to (\mathsf{expand}\ lV\ ((s,n) :: p)).m = (\mathsf{expand}\ lV\ p).m)$
6. $p.i = °(s,n) \to (\forall j < i.\ p.j \neq °(!s,n)) \to n \leq lV \to (\mathsf{expand}\ lV\ p).n = °s$

Proof. 1. By induction on lV.

2. By induction on lV.
3. By induction on lV, doing a case analysis on $m \stackrel{?}{=} 1 + lV$ in the inductive step. Fact 1 is used repeatedly.
4. By repeated case analyses on the result of lookup and using Proposition 29.
5. The second part of the conjunction is statement 4. The first part is proved by induction on lV, doing a case analysis on $0 \stackrel{?}{=} n$ and $m \stackrel{?}{=} 1 + lV$. Statements 1 and 2 are needed.
6. By induction on p, using statements 4 and 5 and doing several case analyses.

◀

▶ **Corollary 31.** *The expanded assignment satisfies the CNF.*

1. $p \prec (1 + lV) \to L \in p \to \mathcal{E}\ (\mathsf{expand}\ lV\ p)\ L\ = {}^\circ\mathsf{T}$
2. $N \prec (1 + lV) \to p \prec (1 + lV) \to \mathcal{E}\ (\mathsf{expand}\ lV\ p)\ N\ = {}^\circ\mathsf{T}$

Proof. 1. By Lemma 30, parts 1 and 6.
2. Directly from part 1 and the assumptions on p.

◀

We now obtain the final result of this subsection:

▶ **Lemma 32.** *Let N : cnf, g : graph and k : N.*

$N \sim^{\boldsymbol{3SAT}}_{\boldsymbol{Clique}} (g, k) \to (\boldsymbol{kSAT}\ 3\ N \leftrightarrow \boldsymbol{Clique}\ (g, k))$

4.4 The Reduction Function

Now that we have shown that it is sufficient to satisfy the reduction relation $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$ in order to get a reduction from **3SAT** to **Clique**, we have to construct a function computing a graph from a CNF obeying this relation and which can be extracted to L and runs in polynomial time.

Writing such a function is straightforward: The main work is generating the graph's edges. We iterate over the literals, from left to right. For each literal, we iterate over all literals in clauses to the right of it[2]. If a pair of literals is non-conflicting, we add an edge. Since the function $\mathsf{makeEdges} : \mathsf{cnf} \to \mathcal{L}\ \mathsf{edge}$ has to keep track of the literal indices, its definition is nevertheless quite technical and we therefore omit it.

The final reduction is given by

$$\mathsf{red}\ N := (\mathbf{if}\ \mathsf{kCNF\_dec}\ 3\ N\ \mathbf{then}\ (3 \cdot |N|, \mathsf{makeEdges}\ N)\ \mathbf{else}\ (0, []), |N|),$$

where kCNF_dec is a Boolean decider for the kCNF predicate. If the input CNF is not valid, we output an empty graph. Since $|N| > 0$ in such a case, the resulting **Clique** instance is not positive, too.

The labelling we use is in line with the above description of makeEdges:

$$labF\ n := (n/3, n \bmod 3) \qquad labF^{-1}\ (cl, l) := 3 \cdot cl + l$$

▶ **Proposition 33.** *Let N be a CNF.*

$\mathsf{isLabelling}\ N\ labF\ labF^{-1}$

Because of the conversion between syntactical literals and literal positions, the verification of makeEdges is very technical. We only state the final result:

▶ **Lemma 34.**

$$\begin{aligned}\mathsf{edgeIn}\ (3 \cdot |N|, \mathsf{makeEdges}\ N)\ a\ b = \mathsf{T} \leftrightarrow \exists\ L, L'.\mathsf{enumLiteral}\ N\ (labF\ a) = {}^\circ L \\ \wedge\ \mathsf{enumLiteral}\ N\ (labF\ b) = {}^\circ L' \wedge \mathsf{diffClause}\ L\ L' \wedge \neg(\mathsf{conflict}\ L\ L')\end{aligned}$$

Now it is easy to show that $\sim^{\mathbf{3SAT}}_{\mathbf{Clique}}$ is satisfied:

▶ **Corollary 35.** $\mathsf{kCNF}\ 3\ N \to N \sim^{\mathbf{3SAT}}_{\mathbf{Clique}} (\mathsf{red}\ N)$

It remains to show that a) red runs in polynomial time and b) the output produced by red has a size polynomial in the size of the input CNF[3]. Both proofs are straightforward.

---

[2] this suffices since we are working with undirected graphs

[3] It does not suffice to show that red runs in polynomial time, since "space bounds time" does not hold for L, in contrast to Turing machines.

5 Running-time Bounds and Coq Formalisation

In this section, we briefly comment on the derivation of the running time bounds. As mentioned in the introduction, the extraction mechanism developed by Forster and Kunze [1] enables us to give running time bounds solving automatically generated recurrences. These running time functions need not be in a closed form, but can be recursive. Moreover, the bounds are not functions of the encoding size of the arguments, but functions of the arguments themselves. In the end, though, we need to show that some function runs in polynomial time in the encoding size of its arguments.

After deriving the time bounds given by the extraction process, we therefore usually prove a lemma providing polynomial time bounds in the encoding size. These lemmas always take the same form. If the function we are analysing is a higher-order function, the time bound will be conditional on a polynomial bound for its argument function. Then we show the existence of $f : \mathsf{N} \to \mathsf{N}$, that, given the sum of the encoding sizes of the function's arguments, computes a bound on the number of steps required, and is polynomial and monotonic. Special care needs to be taken for higher-order functions since the argument function might capture part of the environment, which will then influence the running time.

As an example, we consider the function forallb, checking if a Boolean predicate holds for all elements of a list:

$$\begin{aligned}&\mathsf{forallb}\ (f : A \to \mathsf{B})\ [] := \mathsf{T}\\ &\mathsf{forallb}\ (f : A \to \mathsf{B})\ (l :: ls) := f\ a\ \&\&\ \mathsf{forallb}\ f\ ls\end{aligned}$$

Instantiating the recurrences we get during extraction results in a non-closed running time function which has direct access to the arguments:

$$\mathsf{forallb\_time}\ (fT : A \to \mathsf{N})\ l := \mathsf{foldr}\ (\lambda\ el\ acc.\ fT\ el + acc + 15)\ 8\ l$$

The running time depends on the running time of the Boolean predicate. The appearing constants reflect the number of reduction steps required by the extracted L term.

The running time lemma we then prove is[4]:

$$\begin{aligned}&\forall(fT : E \to A \to \mathsf{N}).(\exists(f : \mathsf{N} \to \mathsf{N}).(\forall el, env.fT\ env\ el \leq f(size(enc\ el) + size(enc\ env)))\\ &\quad \wedge \mathsf{inOPoly}\ f \wedge \mathsf{monotonic}\ f)\\ &\to \exists(f : \mathsf{N} \to \mathsf{N}).(\forall l, env.\ \mathsf{forallb\_time}\ (fT\ env)\ l \leq f(size(enc\ l) + size(enc\ env)))\\ &\quad \wedge \mathsf{inOPoly}\ f \wedge \mathsf{monotonic}\ f\end{aligned}$$

The proofs are usually relatively mechanical, but require an induction on the object the function recurses on, using rewriting with facts about the encoding size of objects. We believe that most of the proofs could be automated, but this would entail a considerable effort for developing suitable procedures.

Some functions need additional assumptions in order to derive bounds. For instance, for foldr we assume that the output size of the aggregation function is only growing by a linear function (not including the accumulator):

$$\exists\, c\, c'.\forall acc, el, env.size(enc(step\ env\ el\ acc)) \leq size(enc\ acc)+c'\cdot(size(enc\ el)+size(enc\ env))+c$$

[4] The definitions of inOPoly and monotonic are just what one would expect them to be.

Even a relaxation to linear functions in the size of the accumulator would not work since the size could then be doubled in each step. The derivation of bounds for tail-recursive functions like foldl is, for this reason, rather unpleasant and requires a non-trivial strengthening of the inductive statement.

6 Closing Remarks

The proof presented in this memo stays conceptually very close to the usual proof on paper, and is rather monolithic. A disadvantage of this technique is that most of the proofs work very closely with the representation of CNFs and graphs. Especially the verification of the makeEdges function is somewhat technical because of the required enumeration of literals and the direct recursion over the CNF when constructing edges.

One reason for the technical overhead is the fact that literals in a CNF are not uniquely identified by their syntax. This required us to work with literal positions and indices. In hindsight, it might have been smarter to reduce **SAT** to an intermediate problem where each literal is annotated with its position, correcting for the lack of uniqueness.

We don't believe the approach taken here will scale well to more complex reductions. Some reductions admit a nice "gadget structure", allowing to build and verify individual gadgets and composition functions. Of course, such a fine-grained approach is not possible for all constructions – for instance, due to the "global" character of the construction of the graph for the reduction to **Clique**, we don't believe that a significantly more clever approach is possible for formalising the construction presented here.

Another path for improvement lies in abstracting away from specific representations and the iteration over these representations. One possible approach would be to use abstract iterator functions taking relations as argument which describe the desired output in a more abstract way. The challenge with such abstractions lies with ensuring extractability to L and nice running time bounds.

References

1 Yannick Forster and Fabian Kunze. A certifying extraction with time bounds from coq to call-by-value lambda-calculus. In *Interactive Theorem Proving - 10th International Conference, ITP 2019, Portland, USA*, Apr 2019. Also available as arXiv:1904.11818.

2 Yannick Forster, Fabian Kunze, and Marc Roth. The weak call-by-value lambda-calculus is reasonable for both time and space. Technical report, Feb 2019. Full version appeared as arXiv:1902.07515 To appear.

3 Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in coq. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasilia, Brazil, September 26-29, 2017*, Apr 2017.