Efficiently Checking Propositional Resolution Proofs in Isabelle/HOL

Tjark Weber[1]
[1]Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
webertj@in.tum.de

Abstract

This paper describes the integration of zChaff and MiniSat, currently two leading SAT solvers, with Isabelle/HOL. Both SAT solvers generate resolution-style proofs for (instances of) propositional tautologies. These proofs are verified by the theorem prover. The presented approach significantly improves Isabelle's performance on propositional problems, and exhibits counterexamples for unprovable conjectures. It is shown that an LCF-style theorem prover can serve as a viable proof checker even for large SAT problems. An efficient representation of the propositional problem in the theorem prover turns out to be crucial; several possible solutions are discussed.

1 Introduction

Interactive theorem provers like PVS [ORS92], HOL [GM93] or Isabelle [Pau94] traditionally support rich specification logics. Proof search and automation for these logics however is difficult, and proving a non-trivial theorem usually requires manual guidance by an expert user. Automated theorem provers on the other hand, while often designed for simpler logics, have become increasingly powerful over the past few years. New algorithms, improved heuristics and faster hardware allow interesting theorems to be proved with little or no human interaction, sometimes within seconds.

By integrating automated provers with interactive systems, we can preserve the richness of our specification logic and at the same time increase the degree of automation [Sha01]. This is an idea that goes back at least to the early nineties [KKS91]. However, to ensure that a potential bug in the automated prover does not render the whole system unsound, theorems in Isabelle, like in other LCF-style [Gor00] provers, can be derived only through a fixed set of core inference rules. Therefore it is not sufficient for the automated prover to return whether a formula is provable, but it must also generate the actual proof, expressed (or expressable) in terms of the interactive system's inference rules.

Formal verification is an important application area of interactive theorem proving. Problems in verification can often be reduced to Boolean satisfiability (SAT), and recent SAT solver advances have made this approach feasible in practice. Hence the performance of an interactive prover on propositional problems may be of significant

practical importance. In this paper we describe the integration of zChaff [MMZ+01] and MiniSat [ES04], two leading SAT solvers, with the Isabelle/HOL [NPW02] prover.

We have shown earlier [Web05a, Web05b] that using a SAT solver to prove theorems of propositional logic dramatically improves Isabelle's performance on this class of formulae, even when a rather naive (and unfortunately, as we will see in Section 3, inefficient) representation of propositional problems is used. Furthermore, while Isabelle's previous decision procedures simply fail on unprovable conjectures, SAT solvers are able to produce concrete counterexamples. In this paper we focus on recent improvements of the proof reconstruction algorithm in Isabelle/HOL, which cause a speedup by several orders of magnitude. In particular the representation of the propositional problem turns out to be crucial for performance. While the implementation in [Web05a] was still limited to relatively small SAT problems, the recent improvements now allow to check proofs with millions of resolution steps in reasonable time. This shows that, somewhat contrary to common belief, efficient proof checking in an LCF-style system is feasible.

The next section describes the integration of zChaff and MiniSat with Isabelle/HOL in more detail. In Section 3 we evaluate the performance of our approach, and report on experimental results. Related work is discussed in Section 4. Section 5 concludes this paper with some final remarks and points out directions for future research.

2 System Description

To prove a propositional tautology ϕ in the Isabelle/HOL system with the help of zChaff or MiniSat, we proceed in several steps. First ϕ is negated, and the negation is converted into an equivalent formula ϕ^* in conjunctive normal form. ϕ^* is then written to a file in DIMACS CNF format [DIM93], the standard input format supported by most SAT solvers. zChaff and MiniSat, when run on this file, return either "unsatisfiable", or a satisfying assignment for ϕ^*.

In the latter case, the satisfying assignment is displayed to the user. The assignment constitutes a counterexample to the original (unnegated) conjecture. When the solver returns "unsatisfiable" however, things are more complicated. If we have confidence in the SAT solver, we can simply trust its result and accept ϕ as a theorem in Isabelle. The theorem is tagged with an "oracle" flag to indicate that it was proved not through Isabelle's own inference rules, but by an external tool. In this scenario, a bug in the SAT solver could potentially allow us to derive inconsistent theorems in Isabelle/HOL.

The LCF-approach instead demands that we verify the solver's claim of unsatisfiability within Isabelle/HOL. While this is not as simple as the validation of a satisfying assignment, the increasing complexity of SAT solvers has before raised the question of support for independent verification of their results, and in 2003 zChaff has been extended by L. Zhang and S. Malik [ZM03] to generate resolution-style proofs that can be verified by an independent checker. (This issue has also been acknowledged by the annual SAT Competition, which has introduced a special track on certified unsat answers in 2005.) More recently, a proof-logging version of MiniSat has been released [ES06], and John Matthews has extended this version to produce human-readable proofs that are easy to parse [Mat06], similar to those produced by zChaff. Hence our main task boils down to using Isabelle/HOL as an independent checker for the resolution proofs

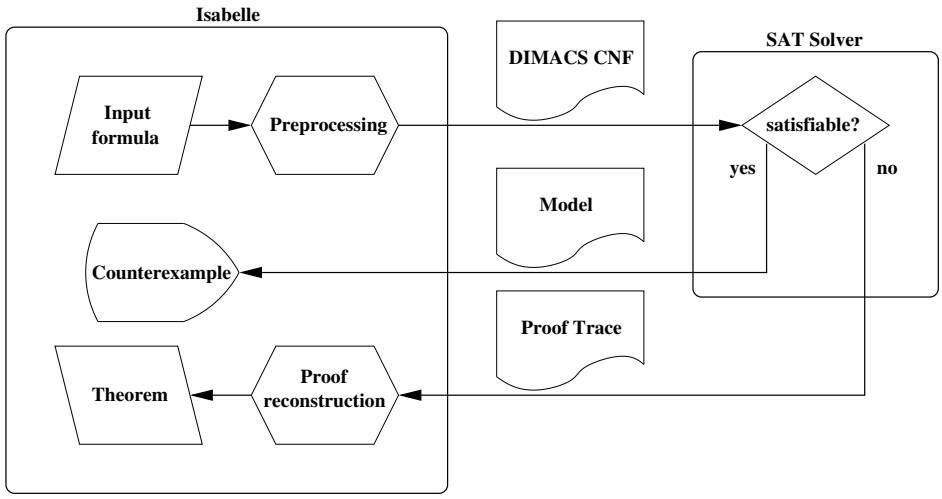

Figure 1: Isabelle – SAT System Architecture

found by zChaff and MiniSat.

Both solvers store their proof in a text file that is read in by Isabelle, and the individual resolution steps are replayed in Isabelle/HOL. Section 2.1 briefly describes the necessary preprocessing of the input formula, and details of the proof reconstruction are explained in Section 2.2. The overall system architecture is shown in Figure 1.

2.1 Preprocessing

Isabelle/HOL offers higher-order logic (on top of Isabelle's meta logic), whereas most SAT solvers only support formulae of propositional logic in conjunctive normal form (CNF). Therefore the (negated) input formula ϕ must be preprocessed before it can be passed to the solver.

Two different CNF conversions are currently implemented: a naive encoding that may cause an exponential blowup of the formula, and a Tseitin-style encoding [Tse83] that may introduce (existentially quantified) auxiliary Boolean variables, cf. [Gor01]. The technical details can be found in [Web05a]. More sophisticated CNF conversions, e.g. from [NRW98], could be implemented as well. The main focus of our work however is on efficient proof reconstruction, less on transformations of the input formula: the benchmark problems used for evaluation in Section 3 are already given in CNF anyway.

Note that it is not sufficient to convert ϕ into an equivalent formula ϕ^* in CNF. Rather, we have to *prove* this equivalence inside Isabelle/HOL. The result is not a single formula, but a theorem of the form $\vdash \phi = \phi^*$. The fact that our CNF transformation must be proof-producing leaves some potential for optimization. One could implement a non proof-producing (and therefore much faster) version of the same CNF transformation, and use it for preprocessing instead. Application of the proof-producing version would then be necessary only if the SAT solver has in fact shown a formula to be unsatisfiable. The total runtime on provable formulae would increase slightly, as the CNF

transformation needed to be done twice – first without, later with proofs. Preprocessing times for unprovable formulae however should improve.

ϕ^* is written to a file in DIMACS CNF format, and the SAT solver is invoked on this input file.

2.2 Proof Reconstruction

When zChaff and MiniSat return "unsatisfiable", they also generate a resolution-style proof of unsatisfiability and store the proof in a text file [ZM03, Mat06]. While the precise format of this file differs between the solvers, the essential proof technique is the same. Both SAT solvers use propositional resolution to derive new clauses from existing ones:

$$\frac{P \vee x \qquad Q \vee \neg x}{P \vee Q}$$

It is well-known that this single inference rule is sound and complete for propositional logic. A set of clauses is unsatisfiable iff the empty clause is derivable via resolution. (For the purpose of proof reconstruction, we are only interested in the proof returned by the SAT solver, not in the techniques and heuristics that the solver uses internally to find this proof. Therefore the integration of zChaff and MiniSat is quite similar – minor differences in their proof trace format aside –, and further SAT solvers capable of generating resolution-style proofs could be integrated with Isabelle in the exact same manner.)

We assign a unique identifier – simply a non-negative integer, starting with 0 – to each clause of the original CNF formula. Further clauses derived by resolution are assigned identifiers by the solver. Often we are not interested in the clause obtained by resolving just two existing clauses, but only in the result of a whole resolution *chain*, where two clauses are resolved, the result is resolved with yet another clause, and so on. Consequently, we define an ML [MTHM97] type of propositional resolution proofs as a pair whose first component is a table mapping integers (to be interpreted as the identifiers of clauses derived by resolution) to lists of integers (to be interpreted as the identifiers of previously derived clauses that are part of the defining resolution chain). The second component of the proof is just the identifier of the empty clause.

```
type proof = int list Inttab.table * int
```

This type is merely intended as an internal format to store the information contained in a resolution proof. There are many restrictions on valid proofs that are not enforced by this type. For example, it does not ensure that its second component indeed denotes the empty clause, that every resolution step is legal, or that there are no circular dependencies between derived clauses. It is only important that every resolution proof can be represented as a value of type `proof`, not conversely. The proof returned by zChaff or MiniSat is translated into this internal format, and passed to the actual proof reconstruction algorithm. This algorithm will either generate an Isabelle/HOL theorem, or fail in case the proof is invalid (which should not happen unless the SAT solver contains a bug).

2.2.1 zChaff's Proof Trace Format

The format of the proof trace generated by zChaff has not been documented before. Therefore a detailed description of the format and its interpretation, although not the main focus of this paper, seems in order.

The proof file generated by zChaff consists of three sections, the first two of which are optional (but present in any non-trivial proof). The first section defines clauses derived from the original problem by resolution. A typical line would be "`CL: 7 <= 2 3 0`", meaning that a new clause was derived by resolving clauses 2 and 3, and resolving the result with clause 0. In this example, the new clause is assigned the identifier 7, which may then be used in further lines of the proof file. Clauses of the original CNF formula are implicitly assigned identifiers starting from 0, in the order they are given in the DIMACS file. When converting zChaff's proof into our internal format, the clause identifiers in a `CL` line can immediately be added to the table which constitutes the proof's first component, with the new identifier as the key, and the list of resolvents as the associated value.

The second section of the proof file records variable assignments that are implied by the first section, and by other variable assignments. As an example, consider "`VAR: 3 L: 2 V: 0 A: 1 Lits: 4 7`". This line states that variable 3 must be false (i.e. its value must be `0`; zChaff uses "`V: 1`" for variables that must be true) at decision level 2, the *antecedent* being clause 1. The antecedent is a clause in which every literal except for the one containing the assigned variable must evaluate to false because of variable assignments at lower decision levels (or because the antecedent is already a unit clause). The antecedent's literals are given explicitly by zChaff, using an encoding that multiplies each variable by 2 and adds 1 for negative literals. Hence "`Lits: 4 7`" corresponds to the clause $x_2 \vee \neg x_3$. Our internal proof format does not allow to record variable assignments directly, but we can translate them by observing that they correspond to unit clauses. For each variable assignment in zChaff's trace, a new clause identifier is generated (using the number of clauses derived in the first section as a basis, and the variable itself as offset) and added as a key to the proof's table. The associated list of resolvents contains the antecedent, and is otherwise obtained from the explicitly given literals: for each literal's variable (except for the one that is being assigned), a similar unit clause must have been added to the table before; its identifier computed according to the same formula. We ignore both the value and the level information in zChaff's trace. The former is implicit in the derived unit clause (which contains the variable either positively or negatively), and the latter is implicit in the overall proof structure.

The last section of the proof file consists of just one line which specifies the *conflict clause*, a clause which has only false literals: e.g. "`CONF: 3 == 4 6`". Literals are encoded in the same way as in the second section, so clause 3 would be $x_2 \vee x_3$ in this case. We translate this line into our internal proof format by generating a new clause identifier i which is added to the proof's table, with the conflict clause itself and the unit clause for each of the conflict clause's variables as associated resolvents. Finally, we set the proof's second component to i.

2.2.2 MiniSat's Proof Trace Format

The proof-logging version of MiniSat originally generated proof traces in a rather compact (and again undocumented) binary format, for which we have not implemented a parser. John Matthews [Mat06] however has extended this version with the ability to produce readable proof traces in ASCII format, similar to those produced by zChaff. We describe the precise proof trace format, and its translation into our internal proof format.

MiniSat's proof traces, unlike zChaff's, are not divided into sections. They contain four different types of statements: "`R`" to reference original clauses, "`C`" for clauses derived via resolution, "`D`" to delete clauses that are not needed anymore, and "`X`" to indicate the end of proof. Aside from "`X`", which must appear exactly once and at the end of the proof trace, the other statements may appear in any number and (almost) any order.

MiniSat does not implicitly assign identifiers to clauses in the original CNF formula. Instead, "`R`" statements, e.g. "`R 0 <= -1 3 4`", are used to establish clause identifiers. This particular line introduces a clause identifier 0 for the clause $\neg x_1 \vee x_3 \vee x_4$, which must have been one of the original clauses in this example. (Note that MiniSat, unlike zChaff, uses the DIMACS encoding of literals in its proof trace.) Since our internal proof format uses different identifiers for the original clauses, the translation of MiniSat's proof trace into the internal format becomes parameterized by a renaming $\mathcal{R}$ of clause identifiers. An "`R`" statement does not affect the proof itself, but it extends the renaming. The given literals are used to look up the identifier of the corresponding original clause, and the clause identifier introduced by the "`R`" statement is mapped to the clause's original (internal) identifier.

New clauses are derived from existing clauses via resolution chains. A typical line would be "`C 7 <= 2 5 3 4 0`", meaning that a new clause with identifier 7 was derived by resolving clauses 2 and 3 (with x_5 as the pivot variable), and resolving the result with clause 0 (with x_4 as the pivot variable). In zChaff's notation, this would correspond to "`CL: 7 <= 2 3 0`". We add this line to the proof's table just like for zChaff, but with one difference: MiniSat's clause identifiers cannot be used directly. Instead, we generate a new internal clause identifier for this line, extend the renaming $\mathcal{R}$ by mapping MiniSat's clause identifier (7 in this example) to the newly generated identifier, and apply $\mathcal{R}$ to the identifiers of resolvents as well.

Clauses that are not needed anymore can be indicated by a "`D`" statement, followed by a clause identifier. Currently such statements are ignored. Making beneficial use of them would require not only a modified proof format, but also a different algorithm for proof reconstruction.

Finally a line like "`X 0 42`" indicates the end of proof. The numbers are the minimum and maximum, respectively, identifiers of clauses used in the proof. We ignore the first identifier (which is usually 0 anyway), and use the second identifier, mapped from MiniSat's identifier scheme to our internal one by applying $\mathcal{R}$, as the identifier of the empty clause, i.e. as the proof's second component.

There is one significant difference between MiniSat's and zChaff's proof traces that should have become apparent from the foregoing description. MiniSat, unlike zChaff, records the pivot variable for each resolution step in its trace, i.e. the variable that occurs

positively in one clause partaking in the resolution, and negatively in the other. This information is redundant, as the pivot variable can always be determined from those two clauses: If two clauses containing more than one variable both positively and negatively were to be resolved, the resulting clause would be tautological, i.e. contain a variable and its negation. Both zChaff and MiniSat are smart enough not to derive such tautological clauses in the first place. We have decided to ignore the pivot information in MiniSat's traces, since proof reconstruction for zChaff requires the pivot variable to be determined anyway, and using MiniSat's pivot data would need a modified internal proof format. This however leaves some potential for optimization wrt. replaying MiniSat proofs.

2.2.3 Proof Reconstruction

We now come to the core of this paper. The task of proof reconstruction is to derive False from the original clauses, using information from a value of type `proof` (which represents a resolution proof found by a SAT solver). This can be done in various ways. In particular the precise representation of the problem as an Isabelle/HOL theorem (or a collection of Isabelle/HOL theorems) turns out to be crucial for performance.

Naive HOL Representation In an early implementation [Web05a], the whole problem was represented as a single theorem $\vdash (\phi^* \Longrightarrow \text{False}) \Longrightarrow (\phi^* \Longrightarrow \text{False})$, where ϕ^* was completely encoded in HOL as a conjunction of disjunctions. Step by step, this theorem was then modified to reduce the antecedent $\phi^* \Longrightarrow \text{False}$ to True, which would eventually prove $\vdash \phi^* \Longrightarrow \text{False}$.

This was extremely inefficient for two reasons. First, every resolution step required manipulation of the whole (possibly huge) problem at once. Second, and just as important, SAT solvers treat clauses as *sets* of literals, making implicit use of associativity, commutativity and idempotence of disjunction. Likewise, CNF formulae are treated as sets of clauses, making implicit use of the same properties for conjunction. The encoding in HOL however required numerous explicit rewrites (with theorems like $\vdash (P \vee Q) = (Q \vee P)$) to reorder clauses and literals before each resolution step.

Separate Clauses Representation A better representation of the CNF formula was discussed in [FMM+06]. In order to understand it, we need to look at the ML datatype of theorems that Isabelle uses internally. Every theorem encodes a *sequent* $\Gamma \vdash \phi$, where ϕ is a single formula, and Γ is a finite set of formulae (implemented as an ordered list of terms, although this detail doesn't matter to us). The intended interpretation is that ϕ holds when every formula in Γ is assumed as a *hypothesis*. So far we have only considered theorems where $\Gamma = \emptyset$, written $\vdash \phi$ for short. This was motivated by the normal user-level view on theorems in Isabelle, where assumptions are encoded using implications $\Longrightarrow$, rather than hypotheses. Isabelle's inference kernel however provides rules that let us convert between hypotheses and implications as we like:

$$\frac{}{\{\phi\} \vdash \phi}\text{Assume} \qquad \frac{\Gamma \vdash \psi}{\Gamma \setminus \phi \vdash \phi \Longrightarrow \psi}\text{impI} \qquad \frac{\Gamma \vdash \phi \Longrightarrow \psi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \psi}\text{impE}$$

Let us use $[\![A_1; \ldots; A_n]\!] \Longrightarrow B$ as a short hand for $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$ (with implication associating to the right). In [FMM+06], each clause $p_1 \vee \ldots \vee p_n$ is encoded

as an implication $\overline{p_1} \Longrightarrow \ldots \Longrightarrow \overline{p_n} \Longrightarrow \text{False}$ (where $\overline{p_i}$ denotes the negation normal form of $\neg p_i$, for $1 \leq i \leq n$), and turned into a separate theorem

$$\{p_1 \vee \ldots \vee p_n\} \vdash [\![\overline{p_1}; \ldots; \overline{p_n}]\!] \Longrightarrow \text{False}.$$

This allows resolution to operate on comparatively small objects, and resolving two clauses $\Gamma \vdash [\![p_1; \ldots; p_n]\!] \Longrightarrow \text{False}$ and $\Gamma' \vdash [\![q_1; \ldots; q_m]\!] \Longrightarrow \text{False}$, where $\neg p_i = q_j$ for some i and j, essentially becomes an application of the cut rule. The first clause is rewritten to $\Gamma \vdash [\![p_1; \ldots; p_{i-1}; p_{i+1}; \ldots; p_n]\!] \Longrightarrow \neg p_i$. A derived Isabelle tactic then performs the cut to obtain

$$\Gamma \cup \Gamma' \vdash [\![q_1; \ldots; q_{j-1}; p_1; \ldots; p_{i-1}; p_{i+1}; \ldots; p_n; q_{j+1}; \ldots; q_m]\!] \Longrightarrow \text{False}$$

from the two clauses. Note that this representation, while breaking apart the given clauses into separate theorems allows us to view the CNF formula as a set of clauses, still does not allow us to view each individual clause as a set of literals. Some reordering of literals is necessary before cuts can be performed, and after each cut, duplicate literals have to be removed from the result.

Sequent Representation We can further exploit the fact that Isabelle's inference kernel treats a theorem's hypotheses as a set of formulae, by encoding each clause using hypotheses *only*. Consider the following representation of a clause $p_1 \vee \ldots \vee p_n$ as an Isabelle/HOL theorem:

$$\{p_1 \vee \ldots \vee p_n, \overline{p_1}, \ldots, \overline{p_n}\} \vdash \text{False}.$$

Resolving two clauses $p_1 \vee \ldots \vee p_n$ and $q_1 \vee \ldots \vee q_m$, where $\neg p_i = q_j$, now starts with two applications of the impI rule to obtain theorems

$$\{p_1 \vee \ldots \vee p_n, \overline{p_1}, \ldots, \overline{p_{i-1}}, \overline{p_{i+1}}, \ldots, \overline{p_n}\} \vdash \neg p_i \Longrightarrow \text{False}$$

and

$$\{q_1 \vee \ldots \vee q_m, \overline{q_1}, \ldots, \overline{q_{j-1}}, \overline{q_{j+1}}, \ldots, \overline{q_m}\} \vdash p_i \Longrightarrow \text{False}.$$

We then instantiate a previously proven theorem

$$\vdash (P \Longrightarrow \text{False}) \Longrightarrow (\neg P \Longrightarrow \text{False}) \Longrightarrow \text{False}$$

(where P is an arbitrary proposition) with p_i for P. Instantiation is another basic operation provided by Isabelle's inference kernel. Finally two applications of impE yield

$$\{p_1 \vee \ldots \vee p_n, \overline{p_1}, \ldots, \overline{p_{i-1}}, \overline{p_{i+1}}, \ldots, \overline{p_n}\} \cup \{q_1 \vee \ldots \vee q_m, \overline{q_1}, \ldots, \overline{q_{j-1}}, \overline{q_{j+1}}, \ldots, \overline{q_m}\} \vdash \text{False}.$$

This approach requires no explicit reordering of literals anymore. Furthermore, duplicate literals do not need to be eliminated after resolution. This is all handled by the inference kernel now; the sequent representation is as close to a SAT solver's view of clauses as sets of literals as possible in Isabelle. With this representation, we do not rely on derived tactics anymore to perform resolution, but we can give a precise description of the implementation in terms of (five, as we see above) applications of core inference rules.

CNF Sequent Representation The sequent representation has the disadvantage that each clause contains itself as a hypothesis. Since hypotheses are accumulated during resolution, this leads to larger and larger sets of hypotheses, which will eventually contain every clause used in the resolution proof. Forming the union of these sets takes the kernel a significant amount of time. It is therefore faster to use a slightly different clause representation, where each clause contains the whole CNF formula ϕ^* as a hypothesis. Let $\phi^* \equiv \bigwedge_{i=1}^{k} C_i$, where k is the number of clauses. Using the Assume rule, we obtain a theorem $\{\bigwedge_{i=1}^{k} C_i\} \vdash \bigwedge_{i=1}^{k} C_i$. Repeated elimination of conjunction (with the help of two theorems, namely $\vdash P \wedge Q \Longrightarrow P$ and $\vdash P \wedge Q \Longrightarrow Q$) yields a list of theorems $\{\bigwedge_{i=1}^{k} C_i\} \vdash C_1$, ..., $\{\bigwedge_{i=1}^{k} C_i\} \vdash C_k$. Each of these theorems is then converted into the sequent form described above, with literals as hypotheses and False as the theorem's conclusion. This representation increases preprocessing times slightly, but throughout the entire proof, the set of hypotheses for each clause now consists of $\bigwedge_{i=1}^{k} C_i$ and the clause's literals only. It is therefore much smaller than before, which speeds up resolution. Furthermore, memory requirements do *not* increase: the term $\bigwedge_{i=1}^{k} C_i$ needs to be kept in memory only once, and can be shared between different clauses. This can also be exploited when the union of hypotheses is formed (assuming that the inference kernel and the underlying ML system support it): a simple pointer comparison is sufficient to determine that both theorems contain $\bigwedge_{i=1}^{k} C_i$ as a hypothesis (and hence that the resulting theorem needs to contain it only once); no lengthy term traversal is required.

We should mention that this representation of clauses, despite its superior practical performance, has a small downside. The resulting theorem always has *every* given clause as a premise, while the theorem produced by the sequent representation only has those clauses as premises that were actually used in the proof. If the logically stronger theorem is needed, it can be obtained by analyzing the resolution proof to identify the used clauses beforehand, and filtering out the unused ones *before* proof reconstruction.

We still need to determine the pivot literal (i.e. p_i and $\neg p_i$ in the above example) before resolving two clauses. This could be done by directly comparing the hypotheses of the two clauses, and searching for a term that occurs both positively and negatively. It turns out to be slightly faster however (and also more robust, since we make fewer assumptions about the actual implementation of hypotheses in Isabelle) to use our own data structure. With each clause, we associate a table that maps integers – one for each literal in the clause – to the Isabelle term representation of a literal. The table is an inverse of the mapping from literals to integers that was constructed for translation into DIMACS format, but restricted to the literals that actually occur in a clause. Positive integers are mapped to positive literals (atoms), and negative integers are mapped to negative literals (negated atoms). This way term negation simply corresponds to integer negation. The table associated with the result of a resolution step is the union of the two tables that were associated with the resolvents, but with the entry for p_i ($\neg p_i$, respectively) removed from the table associated with the first (second, respectively) clause.

Another optimization, related not to the representation of individual clauses, but to the overall proof structure, is perhaps more obvious and has been present in our implementations since the beginning. zChaff and MiniSat, during proof search, may generate many clauses that are ultimately not needed to derive the empty clause. Instead

of replaying the whole proof trace in chronological order, we perform "backwards" proof reconstruction, starting with the identifier of the empty clause, and recursively proving the required resolvents using depth-first search.

While some clauses may not be needed at all, others may be used multiple times in the resolution proof. It would be highly inefficient to prove these clauses over and over again. Therefore all clauses proved are stored in an array, which is allocated at the beginning of proof reconstruction (with a size big enough to possibly hold all clauses derived during the proof). Initially, this array only contains clauses present in the original CNF formula, still in their original format as a disjunction of literals. Whenever an original clause is used as a resolvent, it is converted into the sequent format described above. (Note that this avoids converting original clauses that are not used in the proof at all.) The converted clause, along with its literal table, is stored in the array instead of the original (unconverted) clause. Each clause obtained as the result of a resolution chain is stored as well. Reusing a previously proved clause merely causes an array lookup.

For this reason, it could be beneficial to analyze the resolution chains in more detail: sometimes very similar chains occur in a proof, differing only in a clause or two. Common parts of resolution chains could be stored as additional lemmas (which only need to be derived once), thereby reducing the total number of resolution steps. A detailed evaluation of this idea is beyond the scope of this paper.

2.3 A Simple Example

In this section we illustrate the proof reconstruction using a small example. Consider the following input formula

$$\phi \equiv (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3).$$

Since ϕ is already in conjunctive normal form, preprocessing simply yields the theorem $\vdash \phi = \phi$. The corresponding DIMACS CNF file, aside from its header, contains one line for each clause in ϕ:

```
-1  2 0
-2 -3 0
 1  2 0
-2  3 0
```

zChaff and MiniSat easily detect that this problem is unsatisfiable. zChaff creates a text file with the following data:

```
CL: 4 <= 2 0
VAR: 2 L: 0 V: 1 A: 4 Lits: 4
VAR: 3 L: 1 V: 0 A: 1 Lits: 5 7
CONF: 3 == 5 6
```

We see that first a new clause, with identifier 4, is derived by resolving clause 2, $x_1 \vee x_2$, with clause 0, $\neg x_1 \vee x_2$. The pivot variable which occurs both positively (in clause 2) and negatively (in clause 0) is x_1; this variable is eliminated by resolution.

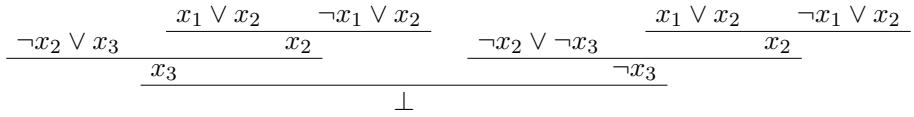

Figure 2: Resolution Proof found by zChaff

Now the value of x_2 (`VAR: 2`) can be deduced from clause 4 (`A: 4`). x_2 must be true (`V: 1`). Clause 4 contains only one literal (`Lits: 4`), namely x_2 (since $4 \div 2 = 2$), occuring positively (since $4 \bmod 2 = 0$). This decision is made at level 0 (`L: 0`), before any decision at higher levels.

Likewise, the value of x_3 can then be deduced from clause 1, $\neg x_2 \vee \neg x_3$. x_3 must be false (`V: 0`).

Finally clause 3 is our conflict clause. It contains two literals, $\neg x_2$ (since $5 \div 2 = 2$, $5 \bmod 2 = 1$) and x_3 (since $6 \div 2 = 3$, $6 \bmod 2 = 0$). But we already know that both literals must be false, so this clause is not satisfiable.

In Isabelle, the resolution proof corresponding to zChaff's proof trace is constructed backwards from the conflict clause. A tree-like representation of the proof is shown in Figure 2. Note that information concerning the level of decisions, the actual value of variables, or the literals that occur in a clause is redundant in the sense that it is not needed by Isabelle to validate zChaff's proof. The clause x_2, although used twice in the proof, is derived only once during resolution (and reused the second time), saving one resolution step in this little example.

The proof trace produced by MiniSat for the same problem happens to encode a different resolution proof:

```
R 0 <= -1 2
R 1 <= -2 -3
R 2 <= 1 2
R 3 <= -2 3
C 4 <= 3 3 1
C 5 <= 0 2 4
C 6 <= 2 2 4
C 7 <= 5 1 6
X 0 7
```

The first four lines introduce clause identifiers for all four clauses in the original problem, in their original order as well (effectively making the renaming $\mathcal{R}$ from MiniSat's clause identifiers to internal clause identifiers the identity in this case). The next four lines define four new clauses (one clause per line), derived by resolution. Clause 4 is the result of resolving clause 3 ($\neg x_2 \vee x_3$) with clause 1 ($\neg x_2 \vee \neg x_3$), where x_3 is used as pivot literal. Hence clause 4 is equal to $\neg x_2$. Likewise, clause 5 is the result of resolving clauses 0 and 4, and clause 6 is obtained by resolving clauses 2 and 4. Finally resolving clauses 5 and 6 yields the empty clause, which is assigned clause identifier 7. The proof is shown in Figure 3. Again one resolution step is saved in the Isabelle implementation because clause $\neg x_2$ is proved only once.