# Efficiently Checking Propositional Refutations in HOL Theorem Provers

Tjark Weber

*Institut für Informatik, Technische Universität München, Boltzmannstr. 3, D-85748 Garching b. München, Germany*

Hasan Amjad

*Computer Laboratory, University of Cambridge, 15 J J Thomson Avenue, Cambridge CB3 0FD, UK*

**Abstract**

This paper describes the integration of zChaff and MiniSat, currently two leading SAT solvers, with Higher Order Logic (HOL) theorem provers. Both SAT solvers generate resolution-style proofs for (instances of) propositional tautologies. These proofs are verified by the theorem provers. The presented approach significantly improves the provers' performance on propositional problems, and exhibits counterexamples for unprovable conjectures. It is also shown that LCF-style theorem provers can serve as viable proof checkers 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.

*Key words:* Interactive Theorem Proving, Propositional Resolution, LCF-style Proof Checking

## 1 Introduction

Interactive theorem provers like PVS [ORS92], HOL4 [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

*Email addresses:* webertj@in.tum.de (Tjark Weber), ha227@cl.cam.ac.uk (Hasan Amjad).

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 integration with the automated prover does not render the whole system unsound, theorems in 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], HOL4 [GM93] and HOL Light [Har96a] theorem provers. All three are LCF-style theorem provers for higher-order logic (HOL), and all use natural deduction as the underlying inference system. The term structure is simple type theory with polymorphism, and the formula syntax is higher-order predicate calculus with equality. Despite these similarities, the underlying implementations of the logic are different enough to pose unique challenges. We shall discuss these in Section 2.4. Outside of that section, we will use tool independent terminology unless explicitly stated otherwise. In particular, we understand a theorem to encode a *sequent* $\Gamma \vdash \phi$, where $\phi$ is a single formula, and $\Gamma$ is a finite set of formulae. The intended interpretation is that $\phi$ holds when every formula in $\Gamma$ is assumed as a *hypothesis*.

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 naïve 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, implemented in the three provers mentioned above, which cause a speedup by several orders of magnitude. In particular the representation of the propositional problem in HOL turns out to be crucial

for performance. While the implementation in [Web05a] was still limited to relatively small SAT problems, our recent improvements now allow us 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 the theorem provers 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 possible directions for future research.

## 2 System Description

To prove a propositional tautology $\phi$ with the help of zChaff or MiniSat, we proceed in several steps. First $\phi$ is negated, and the negation is converted into an equivalent formula $\phi^*$ in conjunctive normal form. $\phi^*$ 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 $\phi^*$.

In the latter case, the satisfying assignment is displayed to the user. The assignment constitutes a counterexample to the original (unnegated) conjecture. Optionally, the satisfying assignment can be substituted into the negation of $\phi$, and a theorem that the counterexample implies the negation of $\phi$ can be derived efficiently by evaluating the resulting ground formula.

When the solver returns "unsatisfiable" however, things are more complicated. We could simply trust accept $\phi$ as a theorem in the HOL theorem prover. The theorem is tagged with an "oracle" flag to indicate that it was proved not through the prover'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. Worse, even if the SAT solver is correct, a bug in the less thoroughly tested translation from HOL to SAT might still render the entire system unsound.

The LCF approach instead demands that we verify the solver's claim of unsatisfiability as well as the correctness of our translation within the prover. While this is not as simple as the validation of a satisfying assignment, the increasing complexity of SAT solvers has before now raised the question of support for independent verification of their results, and in 2003 L. Zhang and S. Malik [ZM03] extended zChaff to generate resolution-style proofs that can be verified by an independent checker. This issue has also been acknowl-

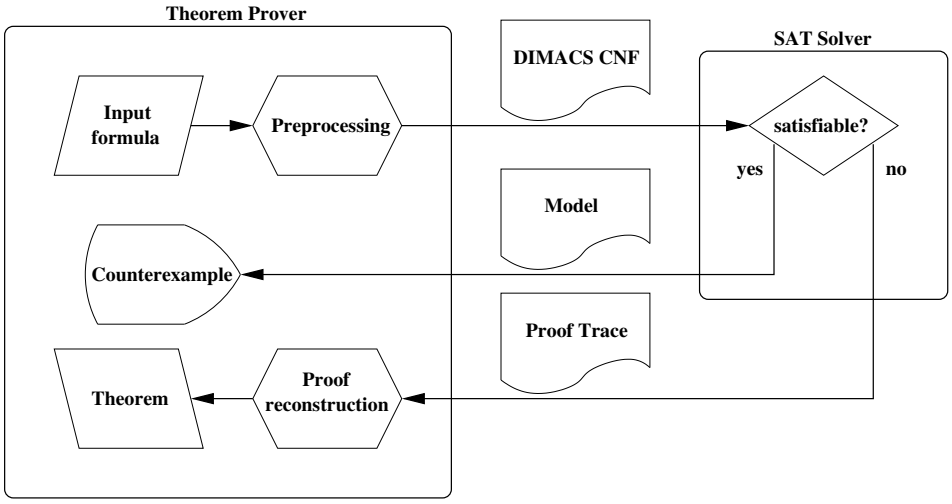


Fig. 1. Theorem Prover – SAT System Architecture

edged by the annual SAT Competition, which introduced a special track on certified "unsatisfiable" answers in 2005. More recently, a proof-logging version of MiniSat has been released [ES06].

From the HOL prover's point of view, using an independent (external) proof checker to verify the SAT solver's answer, while increasing the degree of confidence in the result and likely faster than using the theorem prover for verification, still suffers from potential soundness issues. The independent proof checker, as well as the translation between the different tools, would become part of the trusted code base. Therefore in the LCF framework our main task boils down to using the HOL theorem prover itself as a checker for the resolution proofs found by zChaff and MiniSat.

Both solvers store their proof in a file that is read in by the prover, and the individual resolution steps are replayed using the prover's inference rules. 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*

The provers support arbitrary formulae of propositional logic, whereas most SAT solvers only support formulae of propositional logic in CNF. Therefore the (negated) input formula $\phi$ must be preprocessed before it can be passed to the SAT solver.

Three different CNF conversions are currently implemented: a naïve encoding that may cause an exponential blowup of the formula, an explicit Tseitin-

style "definitional" encoding [Tse83] that avoids exponential blowup but may introduce (existentially quantified) auxiliary Boolean variables, cf. [Gor01], and an implicit Tseitin-style encoding [Bar03] that avoids the quantification and much of the proof. The technical details for the first two can be found in [Web05a]; the third is described below. 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: the benchmark problems used for evaluation in Section 3 are already given in CNF anyway.

Note that it is not sufficient to convert $\phi$ into an equivalent formula $\phi^*$ in CNF. Rather, we have to *prove* this equivalence inside the theorem prover. 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 shown a formula to be unsatisfiable. This scheme can be implemented using lazy proofs [Amj05], thus avoiding the penalty for doing the conversion twice: first without, and later with proofs. This way, preprocessing times for unprovable formulae would improve.

### *2.1.1 Implicit Definitional CNF*

Even the linear blowup of Tseitin-style CNF conversion becomes a bottleneck for large formulas. This is not a concern in benchmarking since SATLIB problems are provided in CNF format. However, it is a serious drawback in promoting everyday use of the system, which is our main motivation. John Harrison suggested an alternative where the definitional variables are not used in logical inferences, but are replaced by their expansions. This is inspired by the approach used for Nelson-Oppen purification in [Bar03], and proceeds as follows:

(1) Use assumption on $\neg\phi$ to obtain $\neg\phi \vdash \neg\phi$.
(2) Split this into several theorems using any top-level conjunctive structure that $\neg\phi$ may have. This is done in a general sense, e.g. top-level terms of the form $p \wedge q$ are obviously split on, but so are $\neg(p \vee q)$, $\neg(p \implies q)$, etc. In addition, double negations are eliminated. This step is then applied recursively to the resulting theorems until no more splits are possible.
(3) The conclusion of each split theorem is converted to CNF without proof, using a standard Tseitin-style CNF encoding.
(4) Then $\phi^*$ is the conjunction of the conclusions of the resulting theorems.

In practice, many problems (including all unsatisfiable SATLIB problems) will have $\phi$ negated at the top-level. Thus, $\neg\phi$ is a double negation, so that step 2

above is more useful than might appear: if $\neg\phi$ were not a double negation, the only situation in which step 2 would do anything at all is if the top-level negation could be converted to a top-level conjunction, e.g. $\neg(p \Rightarrow q)$ becomes $p \wedge \neg q$. When there is no conjunctive structure at the top-level, step 2 terminates.

When replaying the resolution steps in the prover, occurrences of the auxiliary Tseitin variables are substituted by their expansions. This method often avoids doing a Tseitin-style traversal of the entire term, yielding a factor two to five speedup. However, this is not yet implemented in all the provers we consider, so the benchmarks in Section 3 are all for problems that have already been converted to CNF.

Finally, $\phi^*$ 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 generate a resolution-style proof of unsatisfiability and store the proof in a file. The proof trace is logged to the file on-the-fly, to keep memory free for the SAT algorithm itself. While the precise format of this file differs between the solvers, the essential proof structure 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, and further SAT solvers capable of generating resolution-style proofs could be integrated in the same manner.

We assign a unique identifier – a non-negative integer – to each clause of the original CNF formula. Further clauses derived by resolution are assigned unique identifiers by the solver. We are usually interested in the result of a 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 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 a theorem, or fail in case the proof is invalid. Of course the latter should not happen, unless the SAT solver—or our translation from HOL to DIMACS—contains a bug.

### *2.2.1 zChaff Proof Traces*

The format of the proof trace generated by zChaff has not been documented before. We describe it here. We use version 2004.11.15 of zChaff. See Section 2.3 for a simple example of a proof trace.

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, assigned the fresh identifier 7, was derived by resolving clauses 2 and 3, and resolving the result with clause 0. Initial clauses are implicitly assigned identifiers starting from 0, in the order they occur in the DIMACS file.

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`; "`V: 1`" marks true variables) 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 earlier variable assignments (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 us 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 chain of clauses begins with the antecedent, and continues with the unit clauses corresponding to the explicitly given literals. 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 a single line which specifies a clause which has only false literals: e.g. "`CONF: 3 == 4 6`", says clause 3 is $x_2 \vee x_3$ 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 this clause together with the unit clauses for each of the clause's variables forming the chain. Finally, we set the proof's second component to $i$.

For each resolution, we need to determine the pivot literals (i.e. the literals to be resolved on) before resolving two clauses. This could be done by directly comparing 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 clauses in the provers) 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 prover 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 entries for the pivots removed from the tables associated with the two participating clauses.

### *2.2.2 MiniSat Proof Traces*

The proof-logging version of MiniSat generates proof traces in a compact (and again undocumented) binary format. This is mostly likely because SAT competitions currently suggest a limit of 2 GB on proof traces. We use version 1.14p of MiniSat.

MiniSat's proof traces contain three types of statements: original clauses, clauses derived via resolution chains, and deleted clauses. Clause identifiers are implicitly assigned in the order in which the statement appears in the trace, ignoring deletions. Chains use original clauses or earlier derived clauses only. The last chain statement derives the empty clause.

The trace itself is a sequence of numbers occupying 8, 16, 32 or 64 bits. Part

of the first byte of each number encodes this information, so that the parser knows how many more bytes to input before the number is completely read.

An original clause is a list of numbers. Each number is a literal of the clause, using the zChaff encoding except that variable numbering begins at 0. The list is sorted in ascending order with duplicates removed. The beginning of an original clause is indicated by an even number, which needs to be right-shifted by one bit to obtain the number corresponding to the first literal. The remaining literals are recorded as positive adjacent differences, and a zero terminates the clause. Since there are no duplicates, there is no danger of a literal being represented by a zero.

A resolution chain is a list of numbers representing clause identifiers, interleaved with a list of numbers representing the pivot variables. The beginning of a chain statement is indicated by an odd number, which must be right-shifted by one bit and subtracted from the chain's implicit clause identifier (which the parser keeps count of) to yield the first clause identifier in the chain. The next number is the pivot variable and the one after that is another clause identifier, completing the first resolution of the chain. Pivot variable numbers are stored with one added to them (to avoid possible confusion with the terminating zero), and clause identifiers other than the first are stored as positive differences from the chain's clause identifier. As before, a zero terminates the chain.

Deletions are chains where the list of pivot variables is empty, and the single clause identifier is the deleted clause.

MiniSat drops trivial clauses and also performs unit propagation during the read-in phase. Thus the list of original clauses logged (and hence their clause identifiers) may not correspond exactly to the theorem prover's internal list of clauses. We have modified the MiniSat solver to record the original clause identifiers not implicitly by the order in which they appear in the proof trace, but explicitly by the order in which they appear in the original problem. This obviates lookups for original clauses when reading the proof trace. We have also modified MiniSat to record sign information for the pivot. This allows us to dispense with a linear time scan in ML of the literals to determine pivot polarity in the participating clauses.

### *2.3 A Simple Example*

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

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

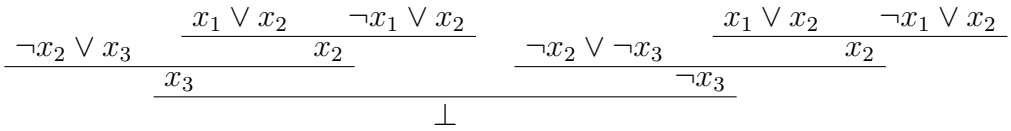


Fig. 2. Resolution Proof found by zChaff

Since $\phi$ 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 $\phi$:

```
-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.

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`).

Clause 3 is our final 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 the prover, 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 to validate zChaff's proof.

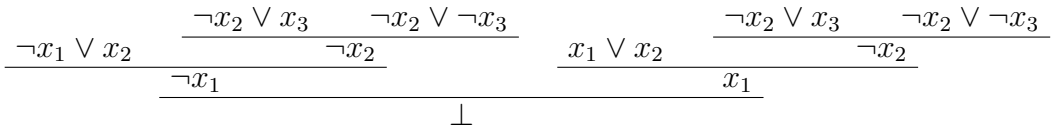


Fig. 3. Resolution Proof found by MiniSat

The proof trace produced by MiniSat for the same problem happens to encode a different resolution proof, shown below in ASCII form, with clause and literal identifiers recovered from the binary encoding:

```
1 2
3 5
0 2
3 4
3 2 1
0 1 4
2 1 4
5 0 6
```

The first four lines introduce the four clauses in the original problem. The next four lines derive four new clauses by resolution, e.g., 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 the pivot variable. Hence clause 4 is equal to $\neg x_2$. The full proof is shown in Figure 3.

### *2.3.1 Proof Trace Compaction*

Before proof reconstruction begins, we can remove redundant and/or unused information from the trace. This can be done without proof, saving time.

**2.3.1.1 Unused and reused derivations** An obvious optimization has been present in all of our implementations. During proof search the SAT solver may derive many clauses that are never used. Since the proof is logged to file on the fly, these derivations end up in the final proof trace. 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 visiting only the required resolvents using depth-first search. We have modified MiniSat to directly output this smaller extracted proof, by doing a disk-based backwards clause traversal on the original trace file. This allows us to read in bigger proofs into memory. Contrary to expectations the disk-based method does not cause a significant performance penalty. We will implement this functionality for zChaff proofs as well.

Some clauses may be used multiple times in the resolution proof. It would be

inefficient to prove these clauses more than once. Therefore clauses are stored in an associative array, keyed on their clause identifier, and upon first use are converted into the sequent format described in Section 2.3.2 below. Reusing a clause is then a single lookup.

**2.3.1.2 Detecting redundant derivations** The above suggests 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. Unfortunately, even if a chain differs from another in a single clause, the end result can be quite different, so simple string matching on chains does not get us much. However, the core idea is not invalidated.

One way to detect redundant computation in chains is to keep track of the clauses that are already known. Then if a resolution step derives a clause that is subsumed by an existing clause, we can skip the resolution and substitute the subsuming clause in its place. This is coupled with unit and binary clause tracking to enable quicker convergence to the empty clause.

To some extent, the clause learning component of SAT solvers tries to avoid redundant search using the same idea. However, it does not target subsumption directly (we know of only one exception [Zha05]), so there is room for improvement.

Our experimental implementation for subsumption based filtering shows promising results in reducing proofs. This implementation is primitive however and currently does not scale beyond toy problems, so it would be premature to provide empirical data at this time. It may be possible to improve performance using the SAT solver's internal data structures, but such an investigation is non-trivial and beyond the scope of this paper.

### *2.3.2 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 a HOL theorem (or a collection of HOL theorems) turns out to be crucial for performance.

**2.3.2.1 Naïve HOL representation** In an early implementation, the whole problem was represented as a single theorem $\vdash (\phi^* \Longrightarrow \text{False}) \Longrightarrow (\phi^* \Longrightarrow \text{False})$, where $\phi^*$ was completely encoded in HOL as a conjunction

of disjunctions [Web05a]. Step by step, this theorem was then modified to reduce the antecedent $\phi^* \Longrightarrow$ False to True, which would eventually prove $\vdash \phi^* \Longrightarrow$ False.

This was extremely inefficient for two reasons. First, every resolution step required manipulation of the whole (possibly huge) problem term 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. Detailed performance figures may be found in [Web05a].

**2.3.2.2 Separate clauses representation** A better representation of the CNF formula was discussed in [FMM+06]. So far we have mostly considered theorems of the form $\vdash \phi$, i.e. with no hypotheses. This was motivated by the normal user-level view of theorems, where assumptions are encoded using implications $\Longrightarrow$ , rather than hypotheses. However, the provers' inference rules 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 $\overline{p_1} \Longrightarrow \ldots \Longrightarrow \overline{p_n} \Longrightarrow$ 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 $\Gamma \vdash [\![p_1; \ldots; p_n]\!] \Longrightarrow$ False and $\Gamma' \vdash [\![q_1; \ldots; q_m]\!] \Longrightarrow$ 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 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.

This representation improved on the proof replay times reported in [Web05a], by up to two orders of magnitude. Detailed numbers can be seen in [FMM+06].

**2.3.2.3 Sequent representation** We can further exploit the fact that the 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 a 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 an implication introduction to obtain

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

then using negation introduction and double negation elimination to get

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

and then using cut (extended to hypotheses) with the sequent representation theorem for the clause $q_1 \vee \ldots \vee q_m$ to yield

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

This approach requires no explicit reordering of literals, nor removal of duplicate literals after resolution. That 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 is possible in a HOL prover. With this representation, we do not rely on derived rules to perform resolution, but can give a precise specification of propositional resolution in terms of a few applications of inference rules of natural deduction. The actual implementation varies depending on the prover.

**2.3.2.4 CNF sequent representation** The sequent representation has the disadvantage that each clause contains itself as a hypothesis. Since hypotheses are accumulated during resolution (specifically, when the cut rule is applied), this leads to larger and larger sets of hypotheses, which will eventually contain every clause used in the resolution proof as an individual term. 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 $\phi^*$ 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 yields a list of theorems $\{\bigwedge_{i=1}^{k} C_i\} \vdash C_1, \ldots, \{\bigwedge_{i=1}^{k} C_i\} \vdash C_k$. Each of these theorems is then converted into the sequent form described above, with literals as hypotheses and False as the theorem's conclusion. Now, throughout the entire proof, the set of hypotheses for each clause consists of a single term $\bigwedge_{i=1}^{k} C_i$ and the clause's literals only. It is therefore much smaller than before, which speeds up resolution.

Furthermore, memory requirements do *not* increase significantly: 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. Thus, even though the size of the sequent using this representation increases in terms of the number of symbols, there is no detrimental effect on either performance or memory use. Note that this representation fits neatly with implicit definitional CNF: we get the required clause representations for free, and only need to perform the contrapositivising step of the sequent representation approach.

We should mention that this representation of clauses, despite its superior 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. To obtain the latter, logically stronger theorem, the resolution proof can be analyzed to identify the clauses that are used in the proof, and the unused ones can be filtered out *before* proof reconstruction.

### *2.4 Prover-specific Issues*

All three of the theorem provers considered support higher-order logic, and all use natural deduction as the inference system. Nonetheless, differences in the implemention of the logic have posed different challenges, which we discuss now.

For us, the relevant differences lie in the data structure representing theorems, the choice of primitive inference rules, and the underlying ML system.

### *2.4.1 Theorem implementation*

Our research highlighted inefficiencies in the kernels of all the provers. These inefficiencies played no important role as long as the kernel only had to deal with relatively small terms, but in our context, where formulae sometimes consist of millions of literals, they turned out to have a measurable negative impact on performance. For example: in Isabelle, instantiating a theorem with a term was linear in the size of the term, rather than constant time; in HOL4, iterated conjunction elimination was not tail-recursive, resulting in a stack overflow; in HOL Light, theorem hypotheses were implemented by unsorted lists, giving quadratic time hypothesis set union. These inefficiencies were removed by the prover developers. In the case of HOL Light, this had a dramatic effect on performance.

The speed of the resolution operation relies critically on being able to form fast unions of the hypothesis sets of theorems. Isabelle/HOL and HOL Light implement hypothesis sets as ordered lists and HOL4 as red-black sets. This means that resolution, asymptotically, is linear in the size of hypotheses in Isabelle/HOL and HOL Light, and $O(n \cdot \log n)$ in HOL4. If the size of the hypotheses set is small, this difference does not play a big role.

However, since there can be millions of resolutions, constant factors become important as the number of inferences rises. Isabelle/HOL resolution may be linear, but has a slightly larger constant factor than HOL4 since implication introduction is linear in Isabelle/HOL, but logarithmic in HOL4. Similarly, HOL Light resolution is also linear, but the current implementation requires one extra traversal of the hypotheses to determine sign information for the pivot. Furthermore, theorems may contain other (prover-dependent) information aside from their hypotheses, e.g. an internal representation of the theorem's proof. This information must be updated by the kernel as well when inferences are performed. These effects show up as differences in performance on checking long proofs with small clauses versus short proofs with big clauses.

We are considering the use of random sets to implement hypotheses. The expected asymptotic performance of these is superior to the solutions outlined above, but the constant factor may be too high.

### *2.4.2 Choice of primitive inference rules*

An LCF-style theorem prover has a kernel of primitive inference rules. Only these rules are allowed direct access to the data structure encoding theorems. All other rules are built by composing the primitives. These *derived* rules may not manipulate theorems directly and are relatively slower as a result.

For our purposes, Isabelle/HOL and HOL4 have all the required inference

rules for simulating propositional binary resolution (assumption, implication introduction, negation introduction, modus ponens) as primitives. HOL4 has a slight advantage in that conjunction elimination, required in the pre-processing stage, is primitive in HOL4 but not in Isabelle/HOL, where the effect is achieved by instantiating pro-forma theorems (namely $\vdash P \wedge Q \Longrightarrow P$ and $\vdash P \wedge Q \Longrightarrow Q$) and using modus ponens. HOL Light has a considerably more primitive kernel. For instance, implication introduction is not primitive but must be derived from the following rule (and others)

$$\frac{\Gamma, p \vdash q \qquad \Delta, q \vdash p}{\Gamma, \Delta \vdash p = q}\text{deduct\_antisym}$$

Even so, it is possible to implement the rule for binary propositional resolution in HOL Light in a manner that is asymptotically as good as that in Isabelle/HOL.

### *2.4.3 Underlying ML system*

HOL4 performance suffers simply because the underlying ML system, Moscow ML, usually runs slower than both Poly/ML (Isabelle/HOL) and OCaml (HOL Light). On the other hand, HOL Light is able to make up for some of its performance loss by being implemented in the fastest of the three ML systems.

Because of all these differences, it is fairly uninformative to directly compare the performance of these provers. None of the three issues above are particularly easy to control for. Thus, while we give performance numbers for all three provers for the record (in Section 3), we caution the reader against inferring any comparative judgements.

## 3 Evaluation

The new sequent representation and improvements to prover implementations enable us to evaluate performance on some significantly harder problems, such as pigeonhole instances and industrial problems taken from the SATLIB library [HS00]. These problems not only push the provers' inference kernels to their limits, but also other components such as the term parser and pretty-printer. For the larger SATLIB problems the parsers (which are mainly intended for small, hand-crafted terms) are unable to parse the problem files, which are several megabytes large, in reasonable time. Also, the prover's user interface is unable to display the resulting formulae. We have therefore implemented our own parser, which builds ML terms directly from DIMACS

| Problem | Vars. | Clauses | Resolutions | zChaff | zChaff+ | zverify_df | Isabelle | HOL4 | HOL Light |
|---|---|---|---|---|---|---|---|---|---|
| c7552mul.miter | 11282 | 69529 | 242509 | 45 | 45 | 1.1 | 69 | 120 | 130 |
| 6pipe | 15800 | 394739 | 310813 | 134 | 137 | 3.7 | 192 | 431 | 1918 |
| 6pipe_6_ooo | 17064 | 545612 | 782903 | 263 | 265 | 5.1 | 421 | 1050 | 3684 |
| 7pipe | 23910 | 751118 | 497019 | 440 | 440 | 6.5 | 609 | 1514 | 6562 |

Table 1
Runtimes (in seconds) for SATLIB problems

| Problem | Vars. | Clauses | Resolutions | zChaff | zChaff+ | zverify_df | Isabelle | HOL4 | HOL Light |
|---|---|---|---|---|---|---|---|---|---|
| pigeon-7 | 56 | 204 | 8705 | <1 | <1 | <0.1 | <1 | 1 | <1 |
| pigeon-8 | 72 | 297 | 25369 | <1 | <1 | 0.1 | 1 | 2 | 1 |
| pigeon-9 | 90 | 415 | 73472 | 1 | 1 | 0.2 | 3 | 5 | 5 |
| pigeon-10 | 110 | 561 | 215718 | 5 | 6 | 0.4 | 10 | 17 | 13 |
| pigeon-11 | 132 | 738 | 601745 | 23 | 24 | 1.2 | 36 | 57 | 57 |
| pigeon-12 | 156 | 949 | 3186775 | 242 | 247 | 6.5 | 315 | 446 | 665 |

Table 2
Runtimes (in seconds) for pigeonhole instances

files, and we work entirely at the systems' ML level, avoiding the usual user interface, to prove unsatisfiability.

For all three provers, statistics for four unsatisfiable SATLIB problems (chosen from those that were used to evaluate zChaff's performance in [ZM03]) are shown in Table 1. Runtimes for selected pigeonhole instances are given in Table 2. The time for zChaff is time taken to solve the problem, without (zChaff) and with (zChaff+) proof logging. (Note that we measure CPU time only, which does not include time spent blocked on I/O. Measuring wall time is pointless because of other processes that may be running simultaneously.) The times for the provers are total times, including zChaff solving time, proof replay, parsing of input and output files, and any other intermediate pre- and post-processing. These timings were obtained on a 1.87 GHz Pentium M notebook with 1.5 GB of main memory. Timings are rounded to the nearest second. For comparison, runtimes for zChaff's own proof checker zverify_df [ZM03] are shown as well, rounded to the nearest tenth of a second. [1]

The proof logging version of MiniSat 1.14 ran out of memory on all problems in Table 1 except c7552mul.miter. This is probably because MiniSat's fast execution tends to find longer proofs, which becomes very costly when proof logging is turned on. The latest version of MiniSat is often better than zChaff (considering the results of the 2006 SAT-Race competition), but unfortunately it did not support proof logging at the time of writing. Therefore we do not give performance data for replaying MiniSat proofs.

Needless to say, none of the SATLIB problems can be solved automatically by the provers' built-in proof procedures. Only the smallest of the pigeonhole instances succumbs, and takes far longer to do so.

Pigeonhole instances are known to be pathologically hard problems for resolu-

[1] The version of zverify_df that comes with zChaff 2004.11.15 contains a minor bug (related to the decision level of variables) which increases its runtime significantly. The above timings were measured with the bug fixed.

tion proof systems. All theorem provers ran out of memory on the pigeonhole problem with 13 holes, even though zChaff found a proof in about 10 minutes. It is hard to do a fine-grained analysis of memory usage, but we can safely say that having to store terms rather than numbers in memory contributed to this failure.

Execution times for Isabelle/HOL are below those for HOL4 and HOL Light. The theorem implementation in Isabelle allows for a smaller constant factor when inferences are performed. A closer look shows that, as discussed in Section 2.4, Isabelle does better when there are fewer but bigger resolutions steps (e.g. problem 7pipe), and HOL4 does better (comparatively) when there are lots of small resolution steps (e.g. problem pigeon-12).

HOL Light is slower than the other provers on SATLIB problems but slightly faster than HOL4 on some pigeonhole instances. We saw in Section 2.4 that HOL Light has a slightly higher constant factor per inference, which becomes worse if the participating clauses are large. This translates to a comparative slowdown on problems which have smaller proofs which involve longer clauses, such as the SATLIB problems. On the pigeonhole problems, the speed of the OCaml system wins out because shorter clauses mean the underlying constant factors per resolution do not dominate.

Proof checking in LCF-style HOL provers, despite all optimizations that we have implemented, is about an order of magnitude slower than proof verification with zChaff's own proof checker zverify_df, written in C++. This additional overhead is to be expected: it is the price that we have to pay for using an LCF-style kernel which is not geared towards propositional logic. However, we also see that proof reconstruction in HOL provers scales quite well with our latest implementation, and that it remains feasible even for large SAT problems.

## 4 Related Work

The work most closely related to ours is John Harrison's LCF-style integration of Stålmarck's algorithm and BDDs into HOL Light and Hol90 (an ancestor of HOL4) respectively [Har96b,Har95]. Harrison found that doing BDD operations inside HOL was about 100 times worse (after several optimizations) than a C implementation. The integration with Stålmarck's algorithm fared much better, but was possibly not folded into the mainstream distribution due to licensing issues.

Mike Gordon implemented *HolSatLib* [Gor01] in Hol98, a precursor to HOL4. This library provided functions to convert Hol98 terms into CNF, and to ana-

lyze them using a SAT solver. In the case of unsatisfiability however, the user only had the option to trust the external solver. No proof reconstruction took place, "since there is no efficient way to check for unsatisfiability using pure Hol98 theorem proving" [Gor01]. A bug in the SAT solver could ultimately lead to an inconsistency in Hol98. The HOL4 implementation of this library is instead based on the work discussed in this paper.

A custom-built SAT solver has been integrated with the CVC Lite system [BB04] by Clark Barrett et al. [BBD03]. The solver produces proofs that can be checked independently. This ability was used to integrate CVC Lite with HOL Light [MBG05]. The integration considerably improved HOL Light's propositional proof capability, but proof reconstruction for CVC Lite's decision procedure for real arithmetic was not fast enough to beat HOL Light's built-in procedure, because of issues similar to the ones we faced when integrating SAT solvers with HOL Light.

Similarly, haRVey, a Satisfiability Modulo Theories (SMT) prover, has been integrated with Isabelle [Hur06]. haRVey, like other SMT systems, uses various decision procedures (e.g. congruence closure for uninterpreted functions) on top of a SAT solver.

Further afield, the integration of automated first-order provers with HOL provers has been explored by Joe Hurd [Hur99,Hur02], Jia Meng [Men03], and Lawrence Paulson [MP04,MP06]. Proofs found by the automated system are either verified by the interactive prover immediately [Hur99], or translated into a proof script that can be executed later [MP04]. Andreas Meier's TRAMP system [Mei00] transforms the output of various automated first-order provers into natural deduction proofs. In the same vein, Mike Gordon and others are working on an integration of the ACL2 prover into HOL4, by embedding the ACL2 logic in HOL and deriving its axioms as HOL4 theorems. The main focus of both however is on the necessary translation from the interactive prover's specification language to first-order logic. In contrast our approach is so far restricted to instances of propositional tautologies, but we have focused on performance (rather than on difficult translation issues), and we use a SAT solver, rather than a first-order prover. Other work on combining proof and model search includes [dNM06].

An earlier version of this work was presented in [Web05a], and improved by Alwen Tiu et al. [FMM+06]. In this paper we have discussed our most recent implementation, which is based on a novel clause representation and constitutes a significant performance improvement when compared to earlier work.

## 5 Conclusions and Future Work

The SAT solver approach dramatically outperforms the automatic procedures that were previously available in the theorem provers we have considered. With the help of zChaff or MiniSat, many formulae that were previously out of the scope of built-in tactics can now be proved–or refuted–automatically, often within seconds. The provers' applicability as a tool for formal verification, where large propositional problems occur in practice, has thereby improved considerably.

Furthermore, using the data structures and optimizations described in this paper, proof reconstruction for propositional logic scales quite well even to large SAT problems and proofs with millions of resolution steps. The additional confidence gained by using an LCF-style prover to check the proof obviously comes at a price (in terms of running time), but it's not nearly as expensive as one might have expected after earlier implementations.

This work is rather new, so there has not as yet been any major verification using it. Some experiments have been performed in the contexts of counterexample driven abstraction refinement and decision procedures for word arithmetic. Our current implementation of the integration has been accepted into the developer repositories of all three provers and will form part of their next release.

While improving the performance of our implementation, we discovered inefficiencies in the implementation of the provers. Subsequently the prover implementations were modified, and these inefficiencies were removed. Tuning an implementation to the extent presented here requires a great deal of familiarity with the underlying theorem prover. Nevertheless our results are applicable beyond these interactive provers, really to any prover that supports propositional logic and is able to simulate propositional resolution.

Roughly speaking, there are four ways one might implement high-performance algorithms for LCF-style provers: trust an external implementation, mimic them in-logic (perhaps extracting a fast executable version if the logic is constructive), reconstruct externally generated proofs, or use reflection to generate efficient proof checkers. The first is unsatisfactory for obvious reasons and the second invariably causes an unacceptable performance penalty. Proof reconstruction works best if the prover's object logic can efficiently simulate the proof system used by the external implementation of the algorithm, or if we can find an efficient translation to such a proof system. This was the case in our work. If it were not so, we can fall back to using some form of reflection, but only as a last resort: reflection is logically tricky and nearly always adds to the trusted code base.

We did not find any soundness bugs in the SAT solvers during proof reconstruction. This is not surprising, since the solvers have already been tested thoroughly on all the problems evaluated above. We did note an odd completeness bug in the verifier bundled with zChaff: it refuses to verify a proof of unsatisfiability if the original problem contains trivial clauses. This is special cased in the code, so the zChaff developers are clearly aware of it. Definitional CNF conversions often generate trivial clauses, so in our setting this is perhaps more important than for the usual verification of unsatisfiable SATLIB problems.

Regarding the proofs produced by SAT solvers, we would like to emphasize the importance of having a well-documented standard, similar to what the DIMACS format is for a SAT solver's input. At present, the mere fact that different solvers use different (and partially undocumented) proof formats makes their integration a bit more of an engineering challenge than it would have to be. Also, solver developers need to be aware that even trivial preprocessing steps (like reordering of clauses) may need to be reproduced in the proof checker. Therefore these steps should (perhaps optionally) be logged in the proof trace as well, or the checker must implement the same preprocessing algorithm as the solver.

We have already mentioned some possible directions for future work. There is probably not very much potential left to optimize the implementation of resolution itself at this point. However, to further improve the performance of proof reconstruction, it could be beneficial to analyze the resolution proof found by the SAT solver in more detail. Merging similar resolution chains may reduce the overall number of resolutions required, and re-sorting resolutions may help to derive shorter clauses during the proof, which should improve the performance of individual resolution steps. Some preliminary results along these lines are reported in [Amj06]. Also preprocessing of CNF formulae for SAT solvers has recently shown very promising results [AS06,EB05], so it might be worthwhile to integrate a preprocessing SAT solver with an LCF-style prover. Note that this is not a trivial task, as the preprocessing must be mimicked inside the HOL prover in a proof-producing fashion.

The approach presented in this paper has applications beyond propositional reasoning. The decision problem for richer logics (or fragments thereof) can be reduced to SAT [ABC$^+$02,Str02,MS05,RH06]. Consequently, proof reconstruction for propositional logic can serve as a foundation for proof reconstruction for other logics. Based on our work, only a proof-generating implementation of the reduction is needed to integrate the more powerful, yet SAT-based decision procedure with an LCF-style theorem prover. Two instances of this approach, using CVC Lite and haRVey, have been mentioned in the previous section.

**Acknowledgments.** Tjark Weber would like to thank Markus Wenzel for

several good ideas and his extensive help with tuning the Isabelle/HOL implementation. The clause representations used in this paper were suggested to him by various people, including John Harrison, John Matthews, and Markus Wenzel.

Hasan Amjad would like to thank John Harrison for help with porting the MiniSat interface to HOL Light, and for developing implicit definitional CNF and suggesting the CNF sequent representation.

Finally both authors would like to thank the anonymous referees for their valuable suggestions.

## References

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 195–210, Copenhagen, Denmark, July 2002. Springer.

[Amj05] Hasan Amjad. Shallow lazy proofs. In Joe Hurd and Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2005.

[Amj06] Hasan Amjad. Compressing propositional refutations. In Stephan Merz and Tobias Nipkow, editors, *Sixth International Workshop on Automated Verification of Critical Systems (AVOCS '06) – Preliminary Proceedings*, pages 7–18, 2006.

[AS06] Anbulagan and John Slaney. Multiple preprocessing for systematic SAT solvers. In C. Benzmüller, B. Fischer, and G. Sutcliffe, editors, *Proceedings of the 6th International Workshop on the Implementation of Logics*, volume 212 of *CEUR Workshop Proceedings*, pages 100–116, Phnom Penh, Cambodia, 2006.

[Bar03] Clark W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, January 2003. Stanford, California.

[BB04] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, Boston, Massachusetts, USA, July 2004.

[BBD03] Clark Barrett, Sergey Berezin, and David L. Dill. A proof-producing Boolean search engine. In *Proceedings of the Workshop on Pragmatics*

*of Decision Procedures in Automated Reasoning (PDPAR 2003)*, Miami, Florida, USA, July 2003.

[DIM93] DIMACS satisfiability suggested format, 1993. Available online at `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc`.

[dNM06] Hans de Nivelle and Jia Meng. Geometric resolution: A proof procedure based on finite model search. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning – Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 2006, Proceedings*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 303–317, 2006.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[ES06] Niklas Eén and Niklas Sörensson. MiniSat-p-v1.14 – A proof-logging version of MiniSat, September 2006. Available online at `www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`.

[FMM+06] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

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

[Gor00] M. J. C. Gordon. From LCF to HOL: A short history. In G. Plotkin, Colin P. Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*. MIT Press, 2000.

[Gor01] M. J. C. Gordon. HolSatLib 1.0b, June 2001. Available online at `http://www.cl.cam.ac.uk/~mjcg/HolSatLib/HolSatLib.html`.

[Har95] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38(2):162–170, 1995.

[Har96a] John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *FMCAD*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[Har96b] John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 1996.

[HS00] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT 2000*, pages 283–292. IOS Press, 2000. Available online at `http://www.satlib.org/`.

[Hur99] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, Nice, France, September 1999. Springer.

[Hur02] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Copenhagen, Denmark, July 2002. Springer.

[Hur06] Clément Hurlin. Proof reconstruction for first-order logic and set-theoretical constructions. In Stephan Merz and Tobias Nipkow, editors, *Sixth International Workshop on Automated Verification of Critical Systems (AVOCS '06) – Preliminary Proceedings*, pages 157–162, 2006.

[KKS91] R. Kumar, T. Kropf, and K. Schneider. Integrating a first-order automatic prover in the HOL environment. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 170–176, Davis, California, USA, August 1991. IEEE Computer Society Press, 1992.

[MBG05] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining CVC Lite and HOL Light. In Alessandro Armando and Alessandro Cimatti, editors, *PDPAR*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Springer, 2005.

[Mei00] Andreas Meier. TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level. In David A. McAllester, editor, *Automated Deduction – CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 460–464. Springer, 2000.

[Men03] Jia Meng. Integration of interactive and automatic provers. In Manuel Carro and Jesus Correas, editors, *Second CologNet Workshop on Implementation Technology for Computational Logic Systems, FME 2003*, September 2003.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, Las Vegas, June 2001.

[MP04] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning: Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 372–384. Springer, 2004.

[MP06] Jia Meng and Lawrence C. Paulson. Translating higher-order problems to first-order clauses. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR: Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, pages 70–80, 2006.

[MS05] Andreas Meier and Volker Sorge. Applying SAT solving in classification of finite algebras. *Journal of Automated Reasoning*, 35(1–3):201–235, October 2005.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, May 1997.

[NPW02] Tobias Nipkow, L. C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NRW98] Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In Claude Kirchner and Hélène Kirchner, editors, *Automated Deduction – CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, volume 1421 of *Lecture Notes in Computer Science*, pages 397–411. Springer, 1998.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[RH06] Erik Reeber and Warren A. Hunt, Jr. A SAT-based decision procedure for the subclass of unrollable list formulas in ACL2 (SULFA). In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning –*

*Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 2006, Proceedings*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 453–467, 2006.

[Sha01] Natarajan Shankar. Using decision procedures with a higher-order logic. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 5–26. Springer, 2001.

[Str02] Ofer Strichman. On solving Presburger and linear arithmetic with SAT. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design: 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*, volume 2517 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2002.

[Tse83] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation Of Reasoning: Classical Papers On Computational Logic, Vol. II, 1967-1970*, pages 466–483. Springer, 1983. Also in *Structures in Constructive Mathematics and Mathematical Logic Part II*, ed. A. O. Slisenko, 1968, pp. 115–125.

[Web05a] Tjark Weber. Integrating a SAT solver with an LCF-style theorem prover. In A. Armando and A. Cimatti, editors, *Proceedings of PDPAR'05 – Third International Workshop on Pragmatical Aspects of Decision Procedures in Automated Reasoning*, Edinburgh, UK, July 2005.

[Web05b] Tjark Weber. Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover. In Joe Hurd, Edward Smith, and Ashish Darbari, editors, *Theorem Proving in Higher Order Logics – 18th International Conference, TPHOLs 2005, Oxford, UK, August 2005, Emerging Trends Proceedings*, pages 180–189, Oxford, UK, August 2005. Oxford University Computing Laboratory, Programming Research Group. Research Report PRG-RR-05-02.

[Zha05] Lintao Zhang. On subsumption removal and on-the-fly CNF simplification. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 482–489. Springer, 2005.

[ZM03] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE 2003)*, pages 10880–10885. IEEE Computer Society, 2003.