LCF-Style Propositional Simplification with BDDs and SAT Solvers

Hasan Amjad

Middlesex University School of Computing Science, London NW4 4BT, UK
Hasan.Amjad@cl.cam.ac.uk

Abstract. We improve, in both a logical and a practical sense, the simplification of the propositional structure of terms in interactive theorem provers. The method uses Binary Decision Diagrams (BDDs) and SAT solvers. We present experimental results to show that the time cost is acceptable.

1 Introduction

We consider the problem of simplifying the propositional structure of terms, in interactive theorem provers (ITPs) based on higher-order logic (HOL) or stronger type systems.

Most such ITPs use rewriting or equational reasoning (semi-automatic or manual) to do such simplification. Such tools include Coq [12], HOL4 [8], HOL Light [10], Isabelle/HOL [18], and PVS [17]. The PVS theorem prover is unique in this family, in additionally optionally allowing the use of Binary Decision Diagrams (BDDs) [1] for propositional simplification [2]. The propositional structure of the input term is encoded as a BDD, from which PVS can automatically extract a term in conjunctive normal form (CNF) that is logically equivalent to the input term. As BDDs usually achieve very compact encodings, the expectation is that the extracted term will be simpler than the input term.

BDDs are powerful tools for propositional reasoning, and have seen widespread adoption in the automated reasoning community. And yet BDDs are rarely if ever used by ITPs. In the current context, there are four broad objections to the use of BDDs for propositional simplification in ITPs:

1. *Not needed*. Current rewriting based implementations of simplifiers are sufficient for the terms that typically occur in interactive proof.
2. *Unsuitable*. The process of BDD-based simplification converts the term to CNF. This destroys the structure of the term and thus may often destroy any intuition that the human ITP user may have had about the term.
3. *Inefficient for LCF-style*. All the ITPs mentioned above, except PVS, are "LCF-style", or follow the "de Bruijn criterion". Roughly speaking, this means that they employ some high assurance facility for verifying their proofs. Typically, this is by translation of the proof to a very simple proof system – the implementation of which is easily understood and well tested

O. Ait Mohamed, C. Muñoz, and S. Tahar (Eds.): TPHOLs 2008, LNCS 5170, pp. 55–70, 2008.
© Springer-Verlag Berlin Heidelberg 2008

– which accepts all correct proofs (and no incorrect ones) produced by the ITP. Verifying BDD operations in this fashion has been tried and found to be very costly [9,21].
4. *Classical.* BDDs are based on classical propositional logic, which limits their usefulness in non-classical contexts.

The last objection cannot be overcome using BDDs as they are currently implemented. Thus, we restrict ourselves to the classical setting. We address the remaining objections in this paper:

1. *Need.* We augment BDD-based simplification with new and known clause-form simplifications and prove that our simplification method can provide logical guarantees about the simplified term that are not provided by current rewrite-based simplifiers. We present experimental results that suggest that in practice our method always does better than rewrite-based simplifiers on a quantifiable measure of simplification quality.
2. *Suitability.* Our method improves on the BDD-based simplification of PVS by not completely flattening the input term. Instead, it selectively applies simplification to suitable sub-terms, thus largely retaining term structure.
3. *Efficiency.* We show that LCF-style BDD-based simplification is possible at not too great a cost, by verifying the BDD operations using a recent LCF-style integration [22] of SAT solvers [16,4] and ITPs. We present experimental results to support this claim.

The next section describes related work. We then give a brief account of the relevant aspects of normal forms, BDDs and SAT solvers, to keep the paper self-contained. Finally, we describe our work (§4) and present experimental results (§5). Henceforth, all discussion is restricted to classical purely propositional HOL terms, unless explicitly stated otherwise.

2 Related Work

There is a reasonable body of work on integrating BDDs in interactive provers. One of the earliest results combined higher-order logic with BDDs for symbolic trajectory evaluation [13]. A little later, temporal symbolic model checking was done in PVS [19]. These integrations trusted the underlying BDD engines. At about the same time, a serious attempt at using BDDs in an LCF-style manner [9] reported an approximate 100x slowdown. Later, a larger project added BDDs to the Coq theorem prover [21] and reported similar slowdowns, except that the faster programs were themselves extracted by reflection from the Coq representation, and could thus said to have higher assurance. The penalty for checking BDD proofs has thus more or less ensured that BDDs are not used internally by LCF-style theorem provers, in a non-trusted manner. There have of course been trusted integrations of BDDs with LCF-style provers [7].

This does not rule out the use of BDDs in interactive provers in general. BDDs are used in the ACL2 prover [14] to help with conditional rewriting and

for deciding equality on bit vectors (see ACL2 System Documentation). The PVS theorem prover can use BDDs for propositional simplification, via its `bddsimp` function [2]. This was the inspiration for our work. Roughly speaking, when invoked on a goal with propositional structure, it uses BDDs to obtain the CNF of the goal, and each conjunct of the CNF becomes a separate subgoal.

LCF-style integrations of SAT solvers with interactive provers have a shorter history. The integration is trivial for the case where the solver returns a satisfying assignment: we simply substitute the assignments into the input term and check that the resulting ground term evaluates to true. This can be done efficiently. For the unsatisfiable case, the earliest work we know of is the LCF-style programming of Stålmarck's Algorithm in an ancestor of HOL4 [11]. This achieved good results but was never distributed due to licensing issues. Further work had to wait for the arrival of DPLL-based proof producing SAT solvers [25] and mature integrations were reported relatively recently [22].

3 Technical Background

We use $\Gamma \vdash t$ to denote that t is a theorem (under hypotheses Γ) in the mechanized object logic, i.e, the logic of the interactive prover. Quantification binds weaker than $\Leftrightarrow$ which binds weaker than all other propositional connectives. Propositional truth is denoted by $\top$ and falsity by $\bot$. All other notation is standard.

In pure propositional logic, there is no concept of variables. In HOL, variables of Boolean type do double duty as propositional letters. We will refer to propositional letters as variables, keeping in mind that quantification over these is not allowed in our setting.

3.1 Normal Forms

A *literal* is either a variable or its negation. Any term has a finite number of variables, so all involved literals can be encoded as numbers when working on a given term. We shall switch between the term and number representation of literals as convenient.

A *clause* is a disjunction of literals. Since both conjunction and disjunction are associative, commutative and idempotent (ACI), clauses can also be interpreted as sets of literals. If a literal occurs in a set, then we abuse notation and assume its underlying proposition also occurs in the set. We assume that any trivial clauses, i.e., containing both a literal and its negation, have been filtered out.

A term is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. Any propositional term t can be transformed into a logically equivalent term in CNF. Again, by ACI, a CNF term can be interpreted as a set (of sets of literals), and we overload the notation accordingly. We will switch back and forth between the term and set interpretations, as convenience dictates.

Any term can be transformed to CNF, but the result can be exponentially larger than the original term. To avoid this, *definitional* CNF [20] introduces extra fresh[1] Boolean variable names as place-holders for subterms of the original

[1] Guaranteed not to already occur in the term.

problem. Conversion to definitional CNF is linear time in the worst case. We use $dCNF(t)$ to denote the definitional CNF of t.

The term $dCNF(t)$ is not logically equivalent to t, since there are valuations for the introduced *definitional variables* that can disrupt an otherwise satisfying assignment to the variables of t. However, it is equisatisfiable. This is expressed by the theorem

$$t \Leftrightarrow \exists V.dCNF(t) \tag{1}$$

where V is the set of all the definitional variables and the existential quantification is lifted to all $v \in V$ in the usual way.

3.2 BDDs

Reduced Ordered Binary Decision Diagrams (ROBDDs, shortened to BDDs) [1] are data structures for efficiently representing Boolean terms and Boolean operations on them. In theory, the problem is NP-complete. In practice, BDDs can often achieve very compact representatations. They are built by starting with the BDDs representing propositional variables and performing a bottom-up construction using the BDD operations corresponding to each propositional connective in the term.

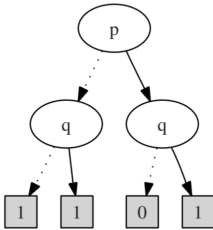

Fig. 1. Binary decision tree for $p \Rightarrow q$

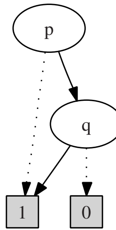

Fig. 2. Corresponding ROBDD

For example, the decision tree of the term $p \Rightarrow q$ is given in Figure 1, and its BDD in 2. Dotted arcs indicate a valuation of $\bot$ and solid arcs a valuation of $\top$ to the parent node. A path from the root to the 1 node indicates an assignment that makes the formula true, and a path to the 0 node, a falsifying assignment.

To read off the CNF equivalent to a term t represented by a BDD, we treat every path from the root to the 0 node, with the signs of variables inverted, as a clause. So, for example, the CNF for Figure 2 is $\neg p \vee q$. This can be done by a single depth-first search of the BDD structure. We denote by $BDDCNF(t)$ the CNF term read off the BDD of a pure propositional term t. We state a standard result:

Proposition 1. $t \Leftrightarrow BDDCNF(t)$

A term is *tautology free* if no strict sub-term of it is true. A term is *contradiction free* if no strict sub-term of it is false. BDDs are canonical and built bottom-up. Hence sub-terms that are tautologies or contradictions are detected during construction, and are absorbed into the BDD. So the following result also holds:

Proposition 2. *BDDCNF*(t) *is tautology free and contradiction free.*

Representing BDDs efficiently in an LCF style prover causes too high a performance penalty (see §2 for details). Therefore, we assume that the results of BDD operations by themselves cannot produce LCF-style theorems in the object logic.

3.3 SAT Solvers

SAT solvers are algorithms for testing Boolean satisfiability. A SAT solver will accept a Boolean term in CNF and return a satisfying assignment to its variables. If the term is unsatisfiable, the solver will simply say so, though some SAT solvers will also return a resolution refutation proof from the clauses of the input CNF term [25]. Such proof-producing SAT solvers have been integrated with LCF-style ITPs [22]. We assume we have access to such an LCF-style integration.

Suppose we have a propositional term t, and we wish to check whether or not it is a tautology. This can be done by computing $dCNF(\neg t)$ (which can be done efficiently) and asking a SAT solver if that term is unsatisfiable. If so, we can derive $\vdash t$.

Thus, we can assume access to a black box procedure $SATprove(t)$ that returns $\vdash t$ iff t is a valid pure propositional term. This short description is sufficient for our purposes. A tutorial introduction to SAT solvers is available [15].

3.4 CNF Simplification

In general, SAT solvers require input in CNF. This has lead to much work on CNF simplification in the SAT community.

Equivalence-preserving CNF simplifications. Of special interest to us are equivalence-preserving simplifications, since these can be directly useful in term simplification where we must derive a logically equivalent but simpler term. Two methods, subsumption reduction (S-reduction for short) and decremental resolution reduction (DR-reduction), have been very effective in practice [3,24].

A clause C *subsumes* another clause D iff $C \subseteq D$, i.e., $C \Rightarrow D$. D is then called *subsumed* and C is called an *S-clause*. Given a clause set, any clauses subsumed by other clauses in the set are redundant and can be removed. This removal, an S-reduction, preserves equivalence, so we have that,

$$\forall CD.(C \Rightarrow D) \Rightarrow (C \wedge D \Leftrightarrow C) \quad (2)$$

A CNF term is considered *subsumption free* (S-free for short) if it has no S-clauses.

The *propositional resolution* rule is

$$\frac{C \vee p \qquad D \vee \neg p}{C \cup D}$$

where the resultant clause is called the *resolvent*, and written as $C \vee p \otimes D \vee \neg p$. p is the *pivot* literal, written as $pivot(C \vee p \otimes D \vee \neg p)$. The pivot occurs

complementarily in the two input clauses; if needed, the sign will be clear from the context.

A resolution is *decremental* if the resolvent implies either of the two input clauses. If so, the implied input clause can be strengthened by removing its pivot literal (equivalent to replacement by the resolvent). This strengthening, a DR-reduction, also preserves equivalence, so we have,

$$\forall CD.(C \otimes D \Rightarrow D) \Rightarrow (C \wedge D \Leftrightarrow C \wedge C \otimes D) \tag{3}$$

A DR-reduction is possible iff one of the two input clauses of the corresponding resolution subsumes the other, modulo the pivot. The almost-subsuming clause is called a *DR-clause*. A CNF term is considered *decremental resolution free* (DR-free for short) if it has no DR-clauses.

We shall adapt the method of Een et al. [3] to our specific circumstances (§4.1). Their method turns a clause set S S-free, by checking whether any $C \in S$ is an S-clause. Let $L(p) = \{C | p \in C \wedge C \in S\}$, i.e, it gives all clauses in which the literal p occurs. Let $\#(C) = \bigoplus_{p \in C} 2^{p \bmod 64}$ where $\oplus$ is bitwise OR, i.e., $\#(C)$ computes a 64-bit hash of clause C. An overview of the algorithm is given in Figure 3, where & is bitwise AND, and ! is bitwise NOT. The test in line 5 is a fast semi-complete subset test: if true it guarantees that $C \not\subseteq D$, and avoids doing the full (and expensive) subset check. They then achieve DR-freedom as shown in Figure 4, where the test in line 3 uses the S-reduction routine.

1. **foreach** $C \in S$
2. $\quad p \leftarrow$ the p such that $p \in C \wedge \forall q \in C. |L(p)| \leq |L(q)|$
3. $\quad$ **foreach** $D \in L(p) - \{C\}$
4. $\quad\quad$ **if** $|C| > |D|$ **then continue**
5. $\quad\quad$ **if** $\#(C) \,\&\, !\#(D) \neq 0$ **then continue**
6. $\quad\quad$ **if** $C \subseteq D$ **then** $S \leftarrow S - \{D\}$

Fig. 3. S-reduction detection

1. **foreach** $C \in S$
2. $\quad$ **foreach** $p \in C$
3. $\quad\quad$ **foreach** $D \in S - \{C\}$ such that $C[p \leftarrow \neg p] \subseteq D$
4. $\quad\quad\quad D \leftarrow D - \{\neg p\}$

Fig. 4. DR-reduction detection

SAT-based CNF simplifications. A considerably more powerful class of CNF simplifications [6,23] removes redundant clauses by using information gleaned from invoking the SAT solver on the CNF term. They are all based on the fact that if a SAT solver produces a refutation from some CNF term t, then clauses of t not participating in the proof can be removed without affecting the

unsatisfiability of t. The same cannot be said if t is satisfiable, since any clauses not used in finding a given satisfying assignment are not necessarily irrelevant to the truth value of t. We shall use the black box call $\mathit{SATsimp}(t)$ to denote the use of such off-the-shelf SAT-based simplifications, under the assumption that the CNF term t is unsatisfiable, and that the call returns some subset c of the clauses of t, such that c is unsatisfiable iff t is unsatisfiable.

4 Simplification

Roughly speaking, the core of our method works as follows:

1. Convert input term t to a BDD and read off the CNF equivalent c_0.
2. Further simplify c_0 using new and known CNF simplifications to obtain c_1.
3. Find redundant clauses in c_1 using SAT-based simplifications to obtain s.
4. If needed, use LCF-style SAT solver interface to prove $\vdash t \Leftrightarrow s$

Note that LCF-style proof is applied only in the final step, so the other phases can be optimised without regard for proof. The main challenges are: avoiding too large a c_0 in BDD construction; fast CNF simplification; using powerful SAT-based simplifications that work only on unsatisfiable terms, for arbitrary terms; achieving a useful term simplification. The last goal is intentionally vague, for now. We discuss this further in §4.3.

4.1 Faster CNF Simplification

The DR-reduction check in Figure 4 requires computing the hash of each $C[p \leftarrow \neg p]$ for each $p \in C$. We cannot compute them incrementally, because we cannot tell whether removing a literal from a clause turned the corresponding bit position in the hash to 0, without considering all the other literals of the clause.

We can improve on this since our goal is simplification of terms encountered in *interactive* proof: in ITPs, automatic or semi-automatic proof procedures very rarely use full-blown simplification internally, for efficiency reasons. This means that the terms we encounter will have considerably fewer variables than the typical SATLIB problem.

Suppose that instead of using a single 64-bit word for the hash, we use enough words so that $\#(-)$ maps each literal to a unique bit position in the hash, i.e., we turn $\#(-)$ into a perfect hashing function. Then the hash-based subset test (Figure 3, line 5) becomes complete. Further, if C is a subset of D modulo some $p \in C$ such that $\neg p \in D$, then $\#(C) \,\&\, !\#(D)$ will have exactly one bit switched on. This can also be detected in constant time (assuming fixed hash size). Figure 5 outlines a method that uses multiword hashes, and combines checking whether some $C \in S$ is an S-clause or a DR-clause. The recursion terminates because the number of literal occurrences in the underlying set S is always strictly smaller with each call. So either eventually there are no more reductions to be found, or we discover the empty clause $\bot$, which subsumes all clauses so S is reduced to $\{\bot\}$. This method is very fast, because it dispenses with the expensive subset checks altogether.

1.$EQsimp_clause(C)$
2. $p \leftarrow$ any p such that $p \in C \wedge \forall q \in C.|L(p)| \leq |L(q)|$
3. **foreach** $D \in L(p) - \{C\}$
4. **if** $|C| > |D|$ **then continue**
5. $h \leftarrow \#(C)\,\&\,!\#(D)$
6. **if** $h = 0$ **then** $S \leftarrow S - \{D\}$
7. **else if** $h\&(h-1) = 0$ **then**
8. $D \leftarrow D - \{pivot(C \otimes D)\}$
9. $EQsimp_clause(D)$

Fig. 5. Combined S-reduction and DR-reduction detection

Our actual implementation is slightly more complex: we also need to check at various points that a clause under consideration has not already been removed from S due to the result of a previous call.

Our overall equivalence-preserving simplification procedure, *EQsimp*, takes as argument a CNF term S (actually a set of set of numbers) and then calls *EQsimp_clause* for each $C \in S$. The number of 64-bit words for the hash is determined once per *EQsimp* call, and is given by dividing the number of variables by 32 and rounding up. This approach is not feasible for SATLIB problems with millions of variables, but in our experiments with interactive goals we rarely needed a hash size of more than four 64-bit words.

Proposition 3. *$EQsimp(t)$ is S-free and DR-free*

Proof $\#(-)$ is now a perfect hash, so bitwise operations on clause hashes coincide with logical operations on clauses. Hence, the test for S-reduction on line 6 of Figure 5 is sound and complete. For DR-reduction, if C subsumes D modulo $pivot(C \otimes D)$, the bit position for the occurrence of the pivot in C will be switched on in both $\#(C)$ and $!\#(D)$, and hence in their conjunction. The remaining bits in the conjunction will be off, as in the S-reduction test. Hence h will be "1-hot", i.e., will have exactly one bit switched on. The test in line 7 turns the most significant switched-on bit of h to off, thereby allowing 1-hot detection. So, the DR-reduction test is sound and complete.

$EQsimp(t)$ checks each clause for being an S-clause or DR-clause. Once checked, a clause cannot again become an S-clause or DR-clause unless it is strengthened in line 8, in which case we recheck it in line 9. Thus, when the algorithm terminates, no clause is an S-clause or a DR-clause. □

Proposition 4. $t \Leftrightarrow EQsimp(t)$

Proof Immediate from (2) and (3). □

4.2 Simplification Using SAT Solvers

Figure 6 gives an overview of our core algorithm, *simplify*, which takes a pure propositional term t and attempts to simplify it by BDD-based conversion to

1. *simplify*(t)
2. $c_0 \leftarrow BDDCNF(t)$
3. $c_1 \leftarrow EQsimp(c_0)$
4. $s \leftarrow SATsimp(c_1 \wedge dCNF(\neg t)) \cap c_1$
5. **return** $SATprove(t \Leftrightarrow s)$

Fig. 6. Core simplification method

CNF, application of CNF simplifications, and SAT-based simplifications, followed by a call to *SATprove* if an LCF-style theorem is required. We have already seen all the simplifications except for SAT-based simplification, which will be the focus here.

We have $t \Leftrightarrow c_1$ by Propositions 1 and 4. However, the powerful *SATsimp* works only for unsatisfiable terms, and in general we cannot expect to be so fortunate. To use it for simplifying arbitrary terms, we devise a specially crafted argument for *SATsimp* and intersect the result with c_1. The correctness of this construction is central to the main theoretical result of the paper.

Theorem 5. $t \Leftrightarrow s$

Proof We have $t \Leftrightarrow c_1$. Now

$$(t \Leftrightarrow c_1) \Leftrightarrow (\neg t \vee c_1) \wedge (\neg c_1 \vee t) \tag{4}$$

and hence

$$c_1 \wedge \neg t \Leftrightarrow \bot \tag{5}$$

Then,

$$\begin{aligned} & c_1 \wedge \neg t \Leftrightarrow \bot \\ \text{iff } & c_1 \wedge (\exists V. dCNF(\neg t)) \Leftrightarrow \bot \quad \text{by (1)} \\ \text{iff } & (\exists V. c_1 \wedge dCNF(\neg t)) \Leftrightarrow \bot \quad \text{no } v \in V \text{ occurs in } c_1 \\ \text{iff } & \forall V. \neg (c_1 \wedge dCNF(\neg t)) \end{aligned}$$

So $c_1 \wedge dCNF(\neg t)$ is unsatisfiable by (5), and of course it is in CNF, meeting the pre-conditions for the call to *SATsimp*, which returns some subset of its input clauses. Clearly $s \subseteq c_1$, hence $c_1 \Rightarrow s$. Then

$$\begin{aligned} & \neg c_1 \wedge t \Rightarrow \bot && \text{by (4) and } t \Leftrightarrow c_1 \\ \text{iff } & \neg s \wedge t \Leftrightarrow \bot && \text{since } \neg s \Rightarrow \neg c_1 \end{aligned} \tag{6}$$

Finally, s is that subset of c_1 that suffices for the proof of $c_1 \wedge dCNF(\neg t) \Leftrightarrow \bot$. So we have a proof of $s \wedge dCNF(\neg t) \Leftrightarrow \bot$. But $s \wedge dCNF(\neg t) \Leftrightarrow \bot$ iff $s \Rightarrow t$ by (1) together with some simple reasoning that uses the fact that no $v \in V$ occurs in s. Combining $s \Rightarrow t$ with (6) gives us $t \Leftrightarrow s$ as required. □

Thus, the call to *SATprove* in line 5 succeeds, and we obtain $\vdash t \Leftrightarrow s$ as desired. Here s obeys the guarantees given by Propositions 2 and 3. Additionally,

we know that s also does not contain any clauses of c_1 not needed by the SAT solver to prove $c_1 \Rightarrow t$. It may nevertheless contain redundant clauses: the solver does not guarantee to use the minimum number of clauses. So we do not phrase this as a guarantee.

4.3 Practicalities

Our hope is that s is "simpler" than t, though we have left this notion vague until now. ITP simplifiers perform many tasks, but in our setting we shall concentrate on the simplest one: given a pure propositional term, what does it mean to simplify it? We believe it means, first, to reduce the size without introducing new operators or variables, and second, to reduce the bracketing depth (modulo associativity) provided this does not conflict too much with the first goal.

These goals are quantifiable and approximate our intuition about propositional simplification to a reasonable degree. The restriction on the second goal implicitly acknowledges that flattening a term too much may lose structure that helps guide intuition. We shall measure term size and bracketing depth in the standard way. We denote the size of term t by $size(t)$.

By these criteria, the *simplify* procedure is too crude. It converts the term to possibly exponentially larger CNF, violating both goals. Instead, we parameterise *simplify* and use it as a subroutine in a control wrapper.

We introduce three parameters B_1, B_2 and F, of which the last two are real numbers, that enforce the following invariants on *simplify*:

1. No BDD has a node count exceeding B_1
2. $size(c_0) < size(t) \times B_2$
3. $size(s) < size(t) \times F$

If any invariant fails, *simplify* signals failure.

The first invariant imposes an upper bound on BDD size, and the second invariant ensures that the CNF generated from the BDD is not too big. The *BDDCNF* function is easily modified to enforce these invariants on-the-fly, rather than checking them after the full BDD or CNF term has been generated. However, our invariant checking for B_1 is not as low-level as it could be: it checks BDD size after each operation, so cannot avoid an exponential size explosion from a single BDD operation. This has not happened yet, but if it becomes a problem, we can simply impose a time limit on the BDD building procedure, rather than a size limit. The third invariant imposes an upper bound on $size(s)$.

All this is not enough, since invoking *simplify* on a term will now invariably report failure. Instead, our control wrapper invokes *simplify* once on every subterm of t in a bottom-up manner. If the invocation succeeds, that sub-term is replaced by its simplified equivalent. Hence, the final simplified term is equivalent to t. Since *simplify* works semantically rather than by non-deterministic rewriting, there is no need to apply it to a given sub-term more than once.

This does mean that the relatively expensive *SATprove* call is made several times. We remedy the situation by modifying *simplify* to return the trivial theorem $t \Leftrightarrow s \vdash t \Leftrightarrow s$ on the last line, which takes negligible cost to generate. Recall

that the control wrapper applies each sub-term simplification to t. So when the bottom up traversal has finished, we have a theorem of the form

$$t_0 \Leftrightarrow s_0, t_1 \Leftrightarrow s_1, \ldots \vdash t \Leftrightarrow s$$

where the t_i are those sub-terms of t which *simplify* succeeded in simplifying to s_i. We now make a single call to $SATprove(\bigwedge_i t_i \Leftrightarrow s_i)$, split the resulting theorem into its conjuncts, and use them to discharge every hypothesis in $t_i \Leftrightarrow s_i \vdash t \Leftrightarrow s$. Even though this *SATprove* call deals with a larger term, in practice these terms are quite small by SAT solver standards, and do not really exercise the SAT interface. Thus, the only LCF-style proof is a single *SATprove* call, followed by very low cost hypothesis discharges that number at most linear in $size(t)$ and considerably fewer in practice.

The guarantees of Proposition 2 and Proposition 3 now only hold locally, at sub-terms where *simplify* succeeded.

As observed earlier, a pleasant side effect of the invariants is that the term structure of t is not flattened beyond the reach of intuition. The degree of flattening can be controlled by the F parameter.

We claimed in the introduction (§1) to simplify the propositional structure of arbitrary terms, rather than pure propositional terms. This is easily done by replacing each atomic proposition with a fresh Boolean variable (but maintaining a bijection between the propositions and the variables), resulting in a pure propositional term that is logically equivalent (modulo the atomic propositions) to the original term. This is not new: PVS already does this, as does the HOL4 decision procedure for propositional tautologies.

As a quick example, consider the term

$$(((p \Rightarrow q) \Rightarrow p) \Rightarrow p) \wedge (p \wedge (p \Leftrightarrow q))$$

The HOL4, HOL Light, and Isabelle/HOL simplifiers (all in default setup) all fail to simplify this term, whereas our method returns $p \wedge q$ as expected. The top-level left conjunct is a tautology (Peirce's Law) that is famously hard for rewriting. The top-level right conjunct is interesting because all three simplifiers fail on it despite its simplicity.[2] Even the *BDDCNF* function by itself returns $p \wedge (\neg p \vee q)$ (assuming p is before q in the BDD ordering) and it requires a DR-reduction to obtain $p \wedge q$. PVS does return $p \wedge q$ but only because after extracting the CNF from the BDD, PVS performs a variant of unit propagation that, as a simplification strategy, is in general both weaker and slower than our reductions.

5 Experimental Results

In §1, we made three claims about the work: that it is not too slow, that it reduces term size better than existing rewrite-based simplifiers, and that it does

[2] They succeed if we make the simplifier aware of certain congruences for conjunction, which are not part of the default set of rewrites because they tend to slow down the rewriter and are not commonly useful.

so without annihilating term structure. Although we have shown that our simplified terms provide certain guarantees of semantic simplicity, these need not always translate into simpler syntax. Hence we present empirical evidence in support of the first two claims.

We compare our method against the HOL4 simplifier. The choice of HOL4 (rather than HOL Light, Isabelle/HOL or PVS) was made because:

1. We wish to evaluate the method in an LCF-style setting, ruling out PVS.
2. The HOL4 SAT interface's definitional CNF subroutine is faster than that of Isabelle/HOL. It avoids all proof by inlining definitions, as opposed to Isabelle/HOL where the definitional CNF produces an expensive LCF-style equivalance proof.
3. We understand the HOL4 simplifier better than the HOL Light simplifier, at the implementation level.

Our method was implemented to produce a valid LCF-style HOL4 theorem, in an interactive HOL4 proof environment. We used a trusted integration of HOL4 with the BuDDy BDD engine [7]. *EQsimp* was implemented by us in C++. For *SATsimp* we used the fixpoint technique of Zhang et al. [23]. For *SATprove*, we used an LCF-style integration of the MiniSat SAT solver [4] with HOL4 [22]. For the comparison, the HOL4 simplifier was invoked using `SIMP_CONV bool_ss []`. The test machine was an AMD Athlon X2 6400+, with 2GB of RAM. Memory consumption was not an issue for these experiments.

We used values of 10000 for B_1, 10.0 for B_2, and 1.1 for F throughout. The B_2 value reflects our intention that the CNF from the BDD should not be too large, but should be big enough to give the CNF simplifications something to work with. The F value reflects our intuition that a "simplified" term that is more than 10% bigger than its original size (without expanding any definitions, which never happens here) is best discarded. We have not experimented with any other values of the parameters.

Since existing propositional benchmark libraries like SATLIB and TPTP (BOO category) have problems already in CNF, we generated random propositional terms for testing, parameterised on a target term size. Randomness meant that for each term size parameter, actual term sizes varied slightly. Nine increasing target sizes were used. For each, the benchmarks were run 100 times. Table 1 presents the minimum, average and maximum term sizes and term bracketing depths for the input and simplified terms, and Table 2 gives the same statistics for execution times, for each size parameter.

As Table 1 shows, our method is always able to achieve a better reduction in term size. In the minimum simplified term sizes for our method, a 1 signifies a random term that was either a tautology or a contradiction, and this was caught by our method as per the guarantee of Proposition 2, and reduced to either $\top$ or $\bot$. HOL4 invariably slightly increases the size of the term. In our method we can control this using the F parameter.

Table 1. Experimental results: term size and bracketing depth

	Term Size									Term Depth								
	Input			HOL4			BDD+SAT			Input			HOL4			BDD+SAT		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
1	7	12.9	16	1	8.4	17	1	2.7	9	1	2.0	3	0	1.3	3	0	0.3	2
2	22	29.5	33	1	26.9	40	1	15.6	33	2	3.5	5	0	3.0	4	0	1.6	4
3	50	59.8	66	1	58.0	74	1	44.2	69	2	4.9	7	0	4.2	6	0	3.8	7
4	70	99.6	122	61	100.2	137	1	75.4	118	4	5.7	7	3	4.8	7	0	4.6	7
5	110	121.3	132	89	124.9	158	1	101.4	127	4	6.2	8	4	5.4	7	0	5.6	8
6	113	125.6	135	62	129.0	149	1	107.1	130	5	6.8	9	4	5.5	8	0	6.3	9
7	161	204.4	233	1	207.0	251	1	171.4	224	5	7.0	9	0	6.1	8	0	6.2	9
8	215	241.7	256	163	249.5	279	1	212.1	246	6	7.7	11	4	6.5	9	0	7.1	9
9	240	251.6	262	205	262.2	289	1	230.5	256	5	8.2	11	5	6.6	8	0	7.7	11

For bracketing depth, HOL4 achieves a marginally better score on larger terms. This may due to inwards movement of negations by HOL4, which reduces depth without badly affecting term size. More detailed analysis of this is needed.

Table 2. Experimental results: execution times (ms)

	HOL4			BDD+SAT		
	min	avg	max	min	avg	max
1	0	0.2	2	10	21.6	58
2	1	1.7	10	12	76.1	266
3	2	4.0	13	16	224.9	944
4	4	6.8	20	20	294.3	589
5	7	11.0	31	25	372.0	637
6	6	9.3	35	27	379.8	553
7	10	14.8	47	33	503.9	847
8	14	17.9	50	42	747.6	1620
9	14	18.7	53	40	876.2	3263

For execution time, (Table 2) HOL4 is about two orders of magnitude faster. However, we are not disappointed. First, in absolute terms our simplifier typically takes less than a second, which is acceptable in interactive proof. Second, our implementation is a mix of SML, C++, Perl and shell scripts, all communicating via disk files, brought on by the need to try out various third-party off-the-shelf implementations. Since the method invokes *simplify* for each subterm, the overhead of file creation, reading, writing, and spawning command shells adds up. This diagnosis is supported by profiling of individual modules: with the exception of *SATsimp*, they contribute very little to the time cost. Our current *SATsimp* works by invoking a SAT solver on the clause set file, extracting only the clauses used by the solver, and repeating the invocation on the new clause set file, until a fixpoint is reached. So it is also disk intensive. We fully expect the execution times to approach those of HOL4 with a little more engineering effort towards direct in-memory interfaces.

6 Conclusion

We have shown how BDDs and SAT solvers can be used for propositional term simplification. The method appears to improve on the simplification of pure propositional terms by rewrite-based simplifiers, at least on random terms with propositional structure sizes the same as those of terms typically encountered in interactive proof. Further we have proved that our simplified terms respect formal guarantees about semantic simplicity that cannot be furnished by rewrite-based simplifiers. Our treatment is tool independent, except that we require that the SAT solver is proof producing. We have also shown that the method can be used in an LCF-style framework with acceptable cost.

We are currently considering a more aggressive structure retention strategy that uses the definitions of definitional CNF to encode structural information, that can later be recovered from the CNF. However, given that the current method does a decent enough job, this is not very high on our list of priorities.

Even though we have restricted ourselves to propositional logic, in theory the results should be applicable to the use of SMT solvers rather than SAT solvers, and should allow us to do simplification in combinations of decidable theories. This awaits the development of a fast and mature LCF-style SMT interface, work on which is underway [5]. If so, our results could then be applied to the propositional structure of more expressive logics via Skolemization, as is done in PVS, since SMT solvers can reason about uninterpreted functions.

Possible direct applications of this work include isolating the cause of a failure to prove a putative tautology, and identification of dead code.[3]

We also plan to use *simplify* in a more fine grained manner, perhaps in conjunction with the rewriting engine of the theorem prover, as is done in ACL2: the HOL4 and Isabelle/HOL simplifiers do provide hooks for integrating other procedures with the rewrite engine. We plan to customize SAT-based simplifiers, for instance to try harder to exclude clauses containing no definitional variables. These plans will form the initial steps for future research.

References

1. Bryant, R.E.: Symbolic boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
2. Cyrluk, D., Rajan, S., Shankar, N., Srivas, M.K.: Effective theorem proving for hardware verification. In: Kumar, R., Kropf, T. (eds.) TPCD 1994. LNCS, vol. 901, pp. 203–222. Springer, Heidelberg (1995)
3. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)

[3] Thanks to Larry Paulson and Laurent Théry respectively, for these ideas.

5. Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L.P., Tiu, A.F.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006)
6. Gershman, R., Koifman, M., Strichman, O.: Deriving small unsatisfiable cores with dominators. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 109–122. Springer, Heidelberg (2006)
7. Gordon, M.J.C.: Programming combinations of deduction and BDD-based symbolic calculation. LMS Journal of Computation and Mathematics 5, 56–76 (2002)
8. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A theorem-proving environment for higher order logic. Cambridge University Press, Cambridge (1993)
9. Harrison, J.: Binary decision diagrams as a HOL derived rule. The Computer Journal 38(2), 162–170 (1995)
10. Harrison, J.: HOL Light. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
11. Harrison, J.: Stålmarck's algorithm as a HOL derived rule. In: von Wright, J., Harrison, J., Grundy, J. (eds.) TPHOLs 1996. LNCS, vol. 1125, pp. 221–234. Springer, Heidelberg (1996)
12. Huet, G., Kahn, G., Paulin-Mohring, C.: The Coq proof assistant: A tutorial: Version 7.2. Technical Report RT-0256, INRIA (February 2002)
13. Joyce, J.J., Seger, C.-J.H.: The HOL-Voss system: Model checking inside a general-purpose theorem prover. In: Joyce, J.J., Seger, C.-J.H. (eds.) HUG 1993. LNCS, vol. 780, pp. 185–198. Springer, Heidelberg (1994)
14. Kaufmann, M., Moore, J.: An industrial strength theorem prover for a logic based on Common Lisp. IEEE Transactions on Software Engineering 23(4), 203–213 (1997)
15. Mitchell, D.G.: A SAT solver primer. In: EATCS Bulletin. The Logic in Computer Science Column, EATCS, vol. 85, pp. 112–133 (February 2005)
16. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, pp. 530–535. ACM Press, New York (2001)
17. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992), `http://pvs.csl.sri.com`
18. Paulson, L.C.: Isabelle. LNCS, vol. 828. Springer, Heidelberg (1994)
19. Rajan, S., Shankar, N., Srivas, M.K.: An integration of model checking and automated proof checking. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 84–97. Springer, Heidelberg (1995)
20. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J., Wrightson, G. (eds.) Automation Of Reasoning: Classical Papers On Computational Logic, Vol. II, 1967-1970, pp. 466–483. Springer, Heidelberg (1983); Slisenko, A.O. (eds.) Structures in Constructive Mathematics and Mathematical Logic Part II, pp. 115–125 (1968)
21. Verma, K.N., Goubault-Larrecq, J., Prasad, S., Arun-Kumar, S.: Reflecting BDDs in Coq. In: He, J., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 162–181. Springer, Heidelberg (2000)
22. Weber, T., Amjad, H.: Efficiently checking propositional refutations in HOL theorem provers. JAL (2007) (accepted for publication, July 2007) (to appear)

23. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formula. In: 6th SAT (2003) (presentation only)
24. Zhang, L.: On subsumption removal and on-the-fly CNF simplification. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 482–489. Springer, Heidelberg (2005)
25. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE, pp. 10880–10885. IEEE Computer Society Press, Los Alamitos (2003)