Proceedings of the

6th International Workshop on the Implementation of Logics

Christoph Benzmüller, Bernd Fischer, Geoff Sutcliffe

Held at

The 13th International Conference on
Logic for Programming, Artificial Intelligence and Reasoning

Phnom Penh, Cambodia
13th - 17th November 2006

The 6th International Workshop on the Implementation of Logics

Christoph Benzmüller[1], Bernd Fischer[2], Geoff Sutcliffe[3]
[1] Universität des Saarlandes, Germany, `chris@ags.uni-sb.de`
[2] University of Southampton, England, `b.fischer@ecs.soton.ac.uk`
[3] University of Miami, `geoff@cs.miami.edu`

The IWIL workshop series brings together developers and users of systems that implement reasoning in logic, to share information about successful implementation techniques for automated reasoning systems and similar programs. Systems of all types (automated, interactive, etc), and for all logics (classical, non-classical, all orders, etc) are of interest to the workshop. Contributions that help the community to understand how to build useful and powerful reasoning systems in practice are of particular interest. Two invited papers, and six research papers selected from the submissions (by the program committee listed below), will be presented. A panel discussion will enable all attendees to participate in open discussion. Previous IWIL workshops include the 5th IWIL (Montevideo, Uruguay), 4th IWIL (Almati, Kazakhstan), 3rd IWIL (Tbilisi, Georgia), 2nd IWIL (Havana, Cuba), and 1st IWIL (Reunion Island).

Journal Publication
The Journal of Algorithms in Applied Logic, Artificial Intelligence and Computer Science has agreed to a special issue based around around the topic of the IWIL workshop. The special issue targets IWIL participants, but will also also accept submissions from the broader community.

Program Committee
Wolfgang Ahrendt (Chalmers University of Technology, Sweden)
Anbulagan (National ICT Australia, Australia)
Serge Autexier (Universität des Saarlandes, Germany)
Chad Brown (Universitt des Saarlandes, Germany)
Hans de Nivelle (Max-Planck Institut fr Informatik, Germany)
Alexander Fuchs (University of Iowa, USA)
Thomas Hillenbrand (Max-Planck Institut für Informatik, Germany)
Boris Konev (University of Liverpool, England)
Konstantin Korovin (University of Manchester, England)
Albert Oliveras (Technical University of Catalonia, Spain)
Brigitte Pientka (McGill University, Canada)
Stephan Schulz (Technische Universität Mnchen, Germany)
Volker Sorge (University of Birmingham, England)
Alwen Tiu (Australian National University, Australia)
Ullrich Hustadt (University of Liverpool, England)

Presentation Schedule

Session 1:	
9:00-10:00	Invited Talk: Algorithms and Data Structures for First-Order Equational Deduction *Stephan Schulz*
Session 2:	
10:30-11:00	Term Indexing for the LEO-II Prover *Frank Theiß and Christoph Benzmüller*
11:00-11:30	Integrating External Deduction Tools with ACL2 *Matt Kaufmann, J Strother Moore, Sandip Ray and Erik Reeber*
11:30-12:00	Efficiently Checking Propositional Resolution Proofs in Isabelle/HOL *Tjark Weber*
Session 3:	
14:00-15:00	Invited Talk: Implementing an Instantiation-based Theorem Prover for First-order Logic *Konstantin Korovin*
Session 4:	
15:30-16:00	Tableau Decision Procedure for Propositional Intuitionistic Logic *Guido Fiorino, Alessandro Avellone and Ugo Moscato*
16:00-16:30	LIFT-UP: Lifted First-Order Planning Under Uncertainty *Steffen Hölldobler and Olga Skvortsova*
16:30-17:00	Multiple Preprocessing for Systematic SAT Solvers *Anbulagan and John Slaney*
17:00-18:00	Panel Interrogation: Experts on the implementation of logics will answer questions from the workshop attendees *Anbulagan, Martin Giese, Konstantin Korovin, Stephan Schulz*

Acknowledgements
We wish to thank the following for their contribution to the success of this conference: Air Force Office of Scientific Research, Asian Office of Aerospace Research and Development.

Algorithms and Data Structures for First-Order Equational Deduction (extended abstract)

Stephan Schulz
Technische Universität München
schulz@eprover.org

1 Introduction

First-order logic with equality is one of the most widely used logics. While there is a large number of different approaches to theorem proving in this logic, the field has been dominated by saturation-based systems using some variant of the superposition calculus [BG90, BG94, BG98, NR01], i.e. systems that employ paramodulation, restricted by ordering constraints and possibly literal selection, as the main inference mechanism, and rewriting and subsumption as the main redundancy elimination techniques. Many systems complement equational reasoning with explicit resolution for non-equational literals. Examples of provers based on the combination of paramodulation, rewriting and (possibly) resolution include SPASS [WBH^{+}02], Vampire [RV01], Otter [MW97], its successor Prover9, and E [Sch02, Sch04b].

The power of a saturating prover depends on four different, but interrelated aspects:

- The calculus (What inferences are necessary and possible?)
- The inference engine (How are they implemented?)
- The search organization (How is the the proof state organized and which invariants are maintained?)
- The heuristic control of the search (Which subset of inferences is performed and in what order?)

In this talk I will discuss the basic concepts of the *given clause* saturation algorithm and its implementation. In particular, I will describe the behaviour of the *DISCOUNT loop* version of this algorithm on practical examples, and discuss how this affected the choice of algorithms and data structures for E. I will try point out some low-hanging fruit, where a lot of performance can be gained for relatively modest investment in code complexity, as well as some more advanced techniques that have a significant pay-off.

2 Rewrite-Based Theorem Proving

Superposition is a refutational calculus. It attempts to make the potential unsatisfiability of a formula (consisting of axioms and negated conjecture) explicit using saturation. The search state is represented by a set of first-order *clauses* (disjunctions of *literals* over *terms*). The proof search employs two different mechanisms. First, new clauses are added to the proof state by *generating inference rules*, using existing clauses as premises. Secondly, *simplifying* or *contracting* inference rules are used to remove clauses or to replace them by simpler ones. The proof is successful if this process eventually produces the *empty clause*, i.e. an explicit contradiction.

While the generating inferences are crucial to establish the theoretical completeness of the calculus, extensive use of contracting inferences has turned out to be indispensable for actually finding proofs.

The practical difficulty of implementing a high-performance theorem prover results mostly from the fact that the proof state grows extremely fast with the depth of the proof search. A successful implementation has to be able to handle large data sets, efficiently find potential inference partners, and in particular, be able to quickly identify clauses that can be simplified or removed.

While the number of inference and simplification rules can be much larger, in nearly all cases only three rules are critical from a performance point of view:

- *Superposition* (including resolution as a special case) or a similar restricted form of paramodulation is by far the most prolific generating inference rule. Typically, between 95% and 99% of clauses in a non-trivial proof search are generated by a paramodulation inference. Paramodulation can essentially be described as a combination of instantiation (guided by unification) and lazy conditional rewriting. For superposition, this inference is further restricted by ordering constraints (a smaller term cannot be replaced by a larger term) and literal selection (only certain literals need to be considered as applicable or as targets of the application).

- Unconditional *rewriting* allows the simplification of a clause by replacing a term with an equivalent, but simpler term. In contrast to superposition, no instantiation of the rewritten clause takes place, and a term is always replaced by a smaller one. As a consequence, this is a *simplifying* inference, and the original of the rewritten clause can be discarded. Practical experience has shown that in most cases rewriting drastically improves the search behaviour of a prover.

- *Subsumption* allows the elimination of clauses if a more general clause already is known. As such, it helps to reduce the search space explosion typical for saturating provers. However, finding subsumption relations between clauses can be very expensive.

For completeness, it is necessary to eventually consider all combinations of non-redundant clauses as premises for generating inferences. To avoid the overhead of keeping track of each possible combination, the *given-clause* algorithms splits the proof state into two distinct subsets, the set P of *processed* (or *active)* clauses, and the set U of *unprocessed* (or *passive*) clauses, and maintains the invariant that all necessary inferences between clauses in P have been performed. On the most abstract level, the algorithm picks a clause from U, performs all inferences with this clause and clauses from P (adding the resulting newly deduced clauses to U), and puts it into P. This process is repeated until either the empty clause is deduced, the set U runs empty (in which case there is not proof), or some time or resource limit has been reached (in which case the prover terminates without a useful result). The major heuristic decision in this algorithm is in which order clauses from U are picked for processing.

The two main variants of the given clause algorithm differ in how they integrate simplification into this basic loop. The *Otter loop* maintains the whole proof state in a fully simplified (or *interreduced)* state. It uses all unit equations for rewriting and all clauses for subsumption attempts. The *DISCOUNT loop*, on which E is built, only maintains this invariant for the set P of processed clauses. It simplifies newly generated clauses once (to weed out obviously redundant clauses and to aid heuristic evaluation, but it does not use them for rewriting or subsumption until they are actually selected for processing. It thus trades reduced simplification for a higher rate of iterations of the main loop. Opinions differ on which of the two designs is more efficient (see e.g. [RV03]).

3 Representing the Proof State

Analysis of the behavior of provers over a large set of examples has shown that the size of U typically grows about quadratically with the size of P. Therefore for non-trivial proof problems, the vast majority of clauses is in U, and unprocessed clauses are responsible for most of the resources used by the prover. The overall proof state can easily reach millions clauses, with a corresponding number of literals and terms as their constituents.

Surprisingly, with naive implementations much of the CPU time can be taken up with seemingly trivial operations. In the first version of DISCOUNT [ADF95, DKS97] we found to our surprise that assumedly complex operations like first-order unification were negligible, while linear time operations like the naive insertion of clauses into U (organized as a linear list sorted by heuristic evaluation) took up around half of the total search time.

The first and most basic data type for a first-order prover is the *term*. Simple, direct implementations realize terms as ordered trees, with each node labeled by a function or variable symbol, and with the subterms of a term as successor nodes in the tree. Alternatives to this basic design are either increasingly optimized for size, as flat-terms, string terms, and eventually implicit terms (reconstructed on demand) as in the Waldmeister prover [GHLS03], or they try to store more and more precomputed information at the term nodes to speed up repetitive operations. This is particularly successful if sets of terms are not represented as sets of trees, but as a shared directed acyclic graph, with repeated occurrences of any given term or subterm only represented once. This approach has been followed in E. We found sharing factors varying from 5-15 in the unit equational case, up to several 1000 in the general non-Horn case.

4 Rewriting

Shared terms do not only allow a reduction in memory consumption, they also allow the sharing of (some) term-related inferences. In particular, they allow the sharing of rewrite operations and, even more importantly, normal form information among terms. For any rewritten term, we can add a link pointing to the result of the rewrite operation. If we encounter the same term again in the future, we can just follow this link instead of performing a real search for matching and applicable rewrite rules. Less glorious, but not less effective, is the sharing of information about non-rewritability. In either version of the given clause algorithm, the set of potential rewrite rules changes over time, and the strength of the rewrite relation grows monotonically. If a term is in normal form with respect to all rules at a given time, no older rule has to be ever considered again for rewriting this term. This criterion can be integrated into indexing techniques to further speed up normal form computation.

5 Subsumption

A single rewriting step is usually cheap to perform. The high cost of rewriting comes from the large number of term positions and rules to consider. For subsumption, on the other hand, even a single clause-clause check can be very expensive, as the problem is known to be NP-complete [KN86]. Unless reasonable care is taken, the exponential worst case can indeed be encountered, and with clauses that are large enough that this hurts performance significantly[1].

To overcome this problem, a number of strategies can be implemented. First, presorting of literals with a suitable ordering stable under substitutions can greatly reduce the number of permutations that need to be considered. Secondly, a number of required conditions for subsumption can be tested rather cheaply. If any of these tests fail, the full subsumption test becomes superfluous.

Feature vector indexing [Sch04a] arranges several of these tests in a way that they can be performed not only for single clauses, but for sets of clauses at a time.

6 Conclusion

Engineering a modern first-order theorem prover is part science, part craft, and part art. Unfortunately, there is no single exposition of the necessary knowledge available at the moment - something that the community should aim to rectify.

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A System for Distributed Equational Deduction. In J. Hsiang, editor, *Proc. of the 6th RTA, Kaiserslautern*, volume 914 of *LNCS*, pages 397–402. Springer, 1995.

[1]This is an understated version of “The prover stops cold”.

[BG90] L. Bachmair and H. Ganzinger. On Restrictions of Ordered Paramodulation with Simplification. In M.E. Stickel, editor, *Proc. of the 10th CADE, Kaiserslautern*, volume 449 of *LNAI*, pages 427—441. Springer, 1990.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[BG98] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (1) of *Applied Logic Series*, chapter 11, pages 353–397. Kluwer Academic Publishers, 1998.

[DKS97] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 2(18):189–198, 1997. Special Issue on the CADE 13 ATP System Competition.

[GHLS03] J.M. Gaillourdet, Th. Hillenbrand, B. Löchner, and H. Spies. The New Waldmeister Loop At Work. In F. Bader, editor, *Proc. of the 19th CADE, Miami*, volume 2741 of *LNAI*, pages 317–321. Springer, 2003.

[KN86] D. Kapur and P. Narendran. NP-Completeness of the Set Unification and Matching Problems. In J.H. Siekmann, editor, *Proc. of the 8th CADE, Oxford*, volume 230 of *LNCS*, pages 489–495. Springer, 1986.

[MW97] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science and MIT Press, 2001.

[RV01] A. Riazanov and A. Voronkov. Vampire 1.1 (System Description). In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 376–380. Springer, 2001.

[RV03] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *Journal of Symbolic Computation*, 36(1–2):101–115, 2003.

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[Sch04a] S. Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving, Cork, Ireland*, ENTCS. Elsevier Science, 2004. (to appear).

[Sch04b] S. Schulz. System Description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[WBH$^+$02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topić. SPASS version 2.0. In A. Voronkov, editor, *Proc. of the 18th CADE, Copenhagen*, volume 2392 of *LNAI*, pages 275–279. Springer, 2002.

Term Indexing for the LEO-II Prover

Frank Theiß[1]
Christoph Benzmüller[2]
[1]FR Informatik, Universität des Saarlandes, Saarbrücken, Germany
lime@ags.uni-sb.de
[2]Computer Laboratory, The University of Cambridge, UK
chris@ags.uni-sb.de

Abstract

We present a new term indexing approach which shall support efficient automated theorem proving in classical higher order logic. Key features of our indexing method are a shared representation of terms, the use of partial syntax trees to speedup logical computations and indexing of subterm occurrences. For the implementation of explicit substitutions, additional support is offered by indexing of bound variable occurrences. A preliminary evaluation of our approach shows some encouraging first results.

1 Introduction

Term indexing has become standard in first order theorem proving and is applied in all major systems in this domain [RV02, Sch02, WBH+02]. An overview on first order term indexing is given in [RSV01] and [NHRV01] presents an evaluation of different techniques. Comparably few term indexing techniques have been developed and studied for higher order logic. An example is Pientka's work on higher order substitution tree indexing [Pie03].

In this paper we present a new approach to higher order term indexing developed for the higher order resolution prover LEO-II[1], the successor of LEO [BK98]. Our approach is motivated by work presented in [TSP06], which studies the application of indexing techniques for interfacing between theorem proving and computer algebra.

Pientka's approach is based on substitution tree indexing and relies on unification of linear higher order patterns. While higher order pattern unification is a comparatively high level operation, the approach we present here is based on coordinate and path indexing [Sti89] and thus relies on lower level operations, for example, operations on hashtables. Apart from indexing and retrieval of terms, we particularly want to speedup basic operations such as replacement of (sub-)terms and occurs checks.

[1]The LEO-II project at Cambridge University has just started (in October 2006). The project is funded by EPSRC under grant EP/D070511/1.

2 Terms in de Bruijn Notation

LEO (and its successor LEO-II under development) is based on Church's simple type theory, that is, a logic built on top of the simply typed λ-calculus [Chu40]. In contrast to LEO, our new term data structure for LEO-II uses de Bruijn [dB72] indices for the internal representation of bound variables. In this paper, de Bruijn indices have the form x_i, where x is a nameless dummy and i the actual index. Constants and free variables in LEO-II, called symbols in the remainder of this paper, have named representations. Due to Currying, applications have only one argument term in LEO-II.[2]

Terms in LEO-II are thus defined as follows:

- *Symbols* are either constant symbols (taken from an alphabet Σ) or (free, existential) variable symbols (taken from an alphabet $\mathcal{V}$). Every symbol is a term.
- *Bound variables*, represented by de Bruijn indices x_i for some index $i \in \{1, 2, \ldots\}$, are terms.
- If s and t are terms, then the *application* $s@t$ is a term.
- If t is a term, then the *abstraction* $\lambda.t$ is a term.

For bound variables x_i, the de Bruijn index i denotes the distance between the variable and its binder in terms of scopes. Scopes are limited by occurrences of λ-binders, thus the index i is determined by the number of occurrences of λ-binders between the variable and its binder.

For instance, the term

$$\lambda a.\lambda b.(b = ((\lambda c.(cb))a))$$

translates to

$$\lambda\lambda.(x_0 = ((\lambda.(x_0 x_1))x_1))$$

in de Bruijn notion. While de Bruijn indices ease the treatment of α-conversion in the implementation, they are less intuitive. As it can be seen in the above example, different occurrences of the same bound variable may have different de Bruijn indices. This is the case here for b, which translates to both x_0 and x_1. Vice versa, different occurrences of the same de Bruijn index may refer to different λ-binders. This is the case for x_0, which relates to both the bound variable b (first occurrence of x_0) and the bound variable c (second occurrence of x_0). Similarly, x_1 is related to the bound variables b and the bound variable a.

[2]Alternative representations, for example, spine notation [CP97], offer at first sight shorter paths to term parts that are relevant for a number of operations. The difference is primarily the order in which the parts of a term can be accessed. In the case of the spine notation, for example, the head symbol of a term can be directly accessed. In our approach we try to offer these shortcuts by representing indexed terms in a graph structure. This allows to adopt additional ways of accessing (sub-)terms by introducing additional graph edges. For instance, the head symbol of each term and its position are indexed in our data structure.

The presentation of LEO-II's type system (simple types) is omitted here. With respect to LEO-II's term indexing, typing only provides an additional criterion for the distinction between terms, for example, different occurrences of the same de Bruijn index may have different types. Apart from this, typing has no further impact on the indexing mechanism. Overall correctness is ensured outside the index structure, that is, by indexing only terms in $\beta\eta$ normal form.

3 The Index

The indexing mechanism we describe here supports fast access to indexed terms and its subterms. The index will be used in LEO-II to store intermediate results (consisting of clauses, literals, and terms in literals) of LEO-II's resolution based proof procedure. The idea is that these intermediate results are always kept in $\beta\eta$ normal form (that is, η short and β normal; for example, the term $(\lambda.(\lambda.(x_1 x_0))))((\lambda.g x_0)c)$ has the $\beta\eta$ normal form (gc)). Hence, $\beta\eta$ normalisation is an invariant of our approach and we assume that we never insert non-normal terms to an index.

Key features of our indexing mechanism are a shared representation of terms (see Section 3.1), the use of partial syntax trees to speedup logical computations (see Section 3.2) and the indexing of subterm occurrences (see Section 3.3). For the implementation of explicit substitutions [FKP96, ACCL90], additional support is offered by indexing of bound variable occurrences (see Section 3.4).

Partial syntax trees are used to index occurrences of symbols and subterms within a term. They help to avoid occurs checks and to early prune superfluous branches in the implementation of operations like replacement, substitution or β-reduction. Checking whether or not a symbol occurs within a term or in a given branch of its syntax tree requires only constant time.

The implementation of the index is furthermore based on the use of *cascaded hashtables*, for example, to index application terms according to their function or argument term. This allows requests for terms in a style similar to SQL [Ame92]. For example, indexing of applications is realised similar to an SQL table `Applications` featuring the columns `appl` for application terms, `func` for their respective function terms and `arg` for their argument terms. Retrieval of terms is similar to SQL queries like "`select appl from Applications where func=t`", which returns terms whose function term is `t`.

3.1 Shared Representation of Terms

Terms in LEO-II have a perfectly shared representation, that is, all occurrences of syntactically equal terms (in de Bruijn notation) are represented by a single instance. An exception are bound variables, where instances of the same variable may have different de Bruijn indices. The treatment of bound variables is described further in Section 3.4.

Terms are represented as *term nodes*. Term nodes are numbered by $n \in \{1, 2, \ldots\}$ in the order they are created. In the following, term nodes are referred to either by their number or by their graph representation, which is defined as follows:

- For each symbol $s \in \Sigma$ occurring in some term, a term node `symbol`(s) is created.
- For each bound variable x_i occurring in some term, a term node `bound`(i) is created.

- If an application $s@t$ occurs in some term, where s is represented by term node i and t by term node j, a term node `application`(i, j) is created.
- If an abstraction λt occurs in some term, where t is represented by term node i, a term node `abstraction`(i) is created.

This graph representation of terms is implemented using hashtables:

- Hashtable `abstr_with_scope` : $I\!N \rightarrow I\!N$ is used to lookup abstractions with a given scope i.
- Hashtable `appl_with_func` : $I\!N \rightarrow I\!N \rightarrow I\!N$ is used to lookup an application with a given function i and argument j.
- Hashtable `appl_with_arg` : $I\!N \rightarrow I\!N \rightarrow I\!N$ is used to lookup an application with a given argument j and function j. This is similar to `appl_with_func`, but the hashtable keys are used here in reversed order.

This hashtable system can be employed to retrieve term nodes in a similar way as in a relational database. It can be used to retrieve single terms as well as sets of terms, for example, all application term nodes whose function term is represented by node i.

3.2 Partial Syntax Trees

Term indexing in LEO-II is based on partial syntax trees (PST), a concept that is newly introduced in this paper. Partial syntax trees are used to indicate positions of a symbol or a particular subterm within a term. PSTs are called *partial* because they only represent *relevant* parts of a term. Examples are PSTs recording symbol occurrences in a term, where relevant part means, that the term part in question actually contains an occurrence of that symbol. Such PSTs allow for early detection of branches in a term's syntax tree with no occurrences of a specific symbol, since these branches are not represented in the PST for this symbol.

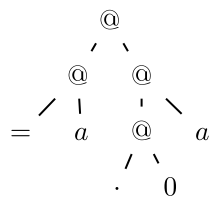
In LEO-II's term system (remember that this is based on simply typed λ-calculus with Currying) a term position is defined as follows:

- While symbol nodes and bound variable nodes have no children in a term's syntax tree, abstraction nodes respectively application nodes have exactly one child respectively exactly two children. The *relative position* of these children to their parent node is described by either `abstr` (the relation between an abstraction node and its scope), `func` or `arg` (the relation between an application node and its function term respectively its argument term).
- A *position* is defined as a (possibly empty) sequence of *relative positions*. Starting from the top position in a term, each entry of the sequence describes one traversal step in the term's syntax tree.
- An empty sequence of *relative positions* represents the *root position* or *empty position*, which is the topmost position in a term.

Consider, for example, the term $(\lambda.x_0)@(f@a)$. Its subterms occur at the following positions:

$(\lambda.x_0)@(f@a)$	:	$[]$
$\lambda.x_0$	:	$[\texttt{func}]$
x_0	:	$[\texttt{func};\texttt{arg}]$
$f@a$	:	$[\texttt{arg}]$
f	:	$[\texttt{arg};\texttt{func}]$
a	:	$[\texttt{arg};\texttt{arg}]$

Based on this notion of positions, we introduce the notion of partial syntax trees. As an example[3], consider the term $a = 0 \cdot a$, which translates in Curried form to $(= @a)@((\cdot@0)@a)$. The example term's syntax tree is given by:



A partial syntax tree (PST) is a tree of nodes corresponding to positions in a term. Each term position which occurs in a PST is represented as a node which

- has up to three child trees[4] (these children are partial syntax trees which correspond to the terms at one of the *relative positions* `abstr`, `func` or `arg`), and
- may be annotated by some data.

A partial syntax tree t is denoted by $pst(t_{abstr}, t_{func}, t_{arg})$, where t_{abstr} is the PST of the scope of t if t is an abstraction, and where t_{func} and t_{arg} are the PSTs of the function term and the argument term of t if t is an application. If no position in a branch of the syntax tree is annotated by some data, this branch is *empty* and is denoted by an underscore (_).

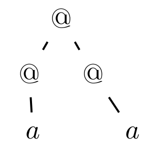
The PST corresponding to the whole term in the above example and its annotations is thus given by:

[3]We present a simple first order example here, since the the treatment of bound variables is special and is described later in Section 3.4.

[4]The data structure of PSTs can not only be *partial*, its structure can also *exceed* the syntax tree of terms as defined in Section 2 if all three children of a node are nonempty. This may be the case when using PSTs to represent *coordinates*, that is term positions which occur in *some* term. This is used when building the index as described in Section 3.3, which is similar to Stickel's path indexing and coordinate indexing methods [Sti89, McC92].

$$\begin{array}{ll}
p_1 = pst(\_, p_2, p_5) & \\
p_2 = pst(\_, p_3, p_4) & \\
p_3 = pst(\_, \_, \_) & \text{with annotation } = \\
p_4 = pst(\_, \_, \_) & \text{with annotation } a \\
p_5 = pst(\_, p_6, p_9) & \\
p_6 = pst(\_, p_7, p_8) & \\
p_7 = pst(\_, \_, \_) & \text{with annotation } \cdot \\
p_8 = pst(\_, \_, \_) & \text{with annotation } 0 \\
p_9 = pst(\_, \_, \_) & \text{with annotation } a
\end{array}$$

When the term is added to the index, however, not the PST of the entire term is recorded, but the PSTs of each of the occurring symbols (and subterms). For example, the PST of all occurrences of the symbol a in $a = 0 \cdot a$ is given by:

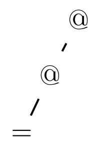


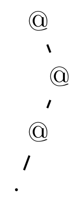
PST for a

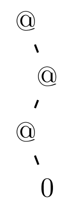
If a symbol occurs at a term position, the corresponding PST entry is annotated by that symbol. If a branch of a term's syntax tree has no occurrences of the symbol in question, the PST contains no entry for this branch. The PST for occurrences of symbol a in the above example is thus given by:

$$\begin{array}{ll}
p_{a1} = pst(\_, p_{a2}, p_{a4}) & \\
p_{a2} = pst(\_, \_, p_{a3}) & \\
p_{a3} = pst(\_, \_, \_) & \text{with annotation } a \\
p_{a4} = pst(\_, \_, p_{a5}) & \\
p_{a5} = pst(\_, \_, \_) & \text{with annotation } a
\end{array}$$

Similarly, the PSTs for the remaining symbols are recorded:







PST for $=$ PST for $\cdot$ PST for 0

If the PST of all occurrences of a symbol (or subterm) t' in a given term t is available, this provides a basis for speeding up replacements of t'. Also a costly occurs check is avoided, since the existence or non-existence of a PST for a symbol can be used as criterion. The nodes of the PST for t' determine the nodes in t that have to be modified when performing the replacement operation, and all nodes in t that are not represented in the PST for t' remain unchanged (i.e., the recursion over the term structure for replacement operations is pruned early).

When replacing a by $(f@b)$ in the above example, the operation proceeds as follows:

- The operations starts at root position with term $(=@a)@((\cdot@0)@a)$ and with the corresponding PST for a, $p_{a1} = pst(\_, p_{a2}, p_{a4})$. As both the function child p_{a2} and the argument child p_{a3} of the PST are nonempty, the replacement operation recurses over both the function term $(=@a)$ and the argument term $((\cdot@0)@a)$:

$$[(f@b)/a](=@a)@((\cdot@0)@a) \Rightarrow ([(f@b)/a](=@a))@([(f@b)/a]((\cdot@0)@a))$$

- To replace a in $(=@a)$ with corresponding PST $p_{a2} = pst(\_,\_,p_{a3})$, only the argument term has to be processed. The child PST corresponding to the function term in p_{a2} is empty, indicating that there are no further occurrences of a in this term. Thus we have:

$$[(f@b)/a](=@a) \Rightarrow (=@([(f@b)/a]a))$$

  and analogously for $((\cdot@0)@a))$ and $p_{a4} = pst(\_,\_,p_{a5})$, where again processing the function term $(\cdot@0)$ is avoided:

$$[(f@b)/a]((\cdot@0)@a)) \Rightarrow ((\cdot@0)@([(f@b)/a]a))$$

- Finally a is replaced in the term a with corresponding PST p_{a3} respectively p_{a5}. Both p_{a3} and p_{a5} have no child nodes and are annotated with a, so the result is in both cases the replacement term $(f@b)$:

$$[(f@b)/a]a \Rightarrow (f@b)$$

The result of this operation is thus:

$$[(f@b)/a](=@a)@((\cdot@0)@a) \Rightarrow (=@(f@b))@((\cdot@0)@(f@b))$$

During the operation, only those branches of the syntax tree with an actual occurrence of a are processed and branches with no occurrences of a, here the terms $=$ and $(\cdot@0)$, are avoided. In this example, only five out of nine term nodes have to be processed due to the guidance provided by the PST.

As a probably useful indicator for the speedup for replacements obtainable this way we therefore investigate the ratio of term size to PST size counted in nodes of the tree, that is, the number of abstractions, applications, symbols and bound variables. As is illustrated above, this ratio is a measure for the speedup which we expect for replacement operations.

In the above example, the term size is 9 (we have 9 nodes), which gives the following rates for the occurring symbols:

Symbol	PST size	PST/term size
a	5	0.56
$=$	3	0.33
$\cdot$	4	0.44
0	4	0.44

We examined an excerpt of Jutting's Automath encoding of Landau's book Grundlagen der Analysis [vBJ77, Lan30] with over 900 definitions and theorems (see Section 4 for details) and found an average PST size/term size rate of 0.21 for symbol occurrences. When indexing nonprimitive terms, too (that is, applications and abstractions), this rate dropped to 0.12.

3.3 Building the Index

The index records whether and at which positions a subterm[5] occurs in a term. Similar to relational databases, both subterms occurring in a given term and terms in which a given subterm occurs are indexed. Thus, the index can be used to find terms in the database with occurrences of particular subterms and also to speed up logical operations such as substitution by avoiding occurs checks.

The index is built recursively, starting from symbols, which are the leaf nodes in a term's syntax tree. The only term which occurs in a symbol is the symbol itself at root position. Nonprimitive terms, that is abstractions and applications are built up as follows:

- The subterms occurring in an *abstraction* are all subterms which occur in the scope of the abstraction. For a symbol whose occurrences in a term A are recorded in the PST t', its occurrences in the abstraction $\lambda.A$ are given by $t = pst(t', \_, \_)$.
- The subterms occurring in an *application* are all subterms which occur in its function term or in its argument term. For a symbol whose occurrences in the function and argument term are recorded in the PSTs t'_{func} and t'_{arg}, the PST t recording its occurrences in the application is given by $t = pst(\_, t'_{func}, t'_{arg})$. If the term occurs only in the argument respectively in the function of an application, t_{func} respectively t_{arg} is *empty*.

Furthermore each primitive and nonprimitive term is recorded to occur as a subterm of itself at root position.

The result is a PST for each subterm of the term to be indexed, describing the occurrences of this subterm. These PSTs are added to the hashtable `occurrences`. Additionally, terms are indexed according to their subterms in a second hashtable `occurs_in`. A third hashtable is `occurrs_at`, which is used to index terms according to subterms at a given term position. Thus the core of the index consists of:

[5]In the current implementation, both occurring symbols and nonprimitive subterms are indexed. However, we plan to further evaluate the tradeoff between the speedup gained this way and the cost for maintenance of the index. Depending on this evaulation, we may want to restrict the nonprimitive subterms to be indexed, for example, by using their size as a criterion. Similar ideas have been examined by McCune [McC92], for example, the effect of limitations on the length of paths used in path indexing.

- Hashtable `occurrences` : $I\!N \rightarrow I\!N \rightarrow PST$ indexes occurrences of subterms (the second key) in a given term (the first key). The indexed value is a PST of the positions where the subterm occurs. If a subterm does not occur, then there is no entry in the hashtable.

- Hashtable `occurs_in` : $I\!N \rightarrow I\!N^*$ is used to index a list of all terms in which a given subterm (the key) occurs.

- Hashtable `occurrs_at` : $pos \rightarrow I\!N \rightarrow I\!N^*$ is a hashtable to index all terms in which a given subterm (the second key) occurs at a given position (the first key).

For example, occurrences of symbol a in the example term $(= @a)@((\cdot@0)@a)$ are indexed by the following hashtable updates (we assume that a is represented by term node i and $(= @a)@((\cdot@0)@a)$ by term node j):

- add pst_a with first key j and second key i in `occurrences`

- add j with key i in `occurs_in`

- add j to the set hashed in `occurs_at` with first key [`func`; `arg`] and second key i; if no such set exists in the hashtable, add the singleton $\{j\}$

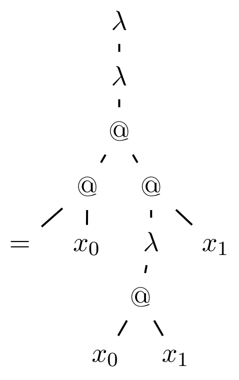
- add j to the set hashed in `occurs_at` with first key [`arg`; `arg`] and second key i; if no such set exists in the hashtable, add the singleton $\{j\}$

The basic operations of adding a term to the index take constant time (except for rehashing). The indexing of a term of length n takes time $O(n)$.

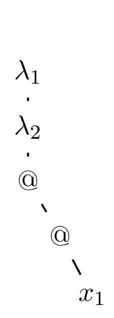
3.4 Bound Variables

Bound variables play a special role in the term system. To see this, remember our example from the beginning, that is, the term $\lambda a.\lambda b.((= b)((\lambda c.(cb))a))$ or, with de Bruijn indices, $\lambda\lambda.((= x_0)((\lambda.(x_0x_1))x_1))$. This example shows that two occurrences of the same bound variable may have syntactically different de Bruijn indices and that the de Bruijn indices of occurrences of different variables may be syntactically equal. It is desirable to provide quick access to all variables bound by a given binder to speedup β-reduction and related operations such as raising or lowering of bound variable indices. We will now illustrate our solution to this issue. Remember that indexed terms are always kept in $\beta\eta$ normal form, hence, normalisation is mandatory after instantiation of existential variables or expansion of defined constants (if the modified terms shall be indexed again).

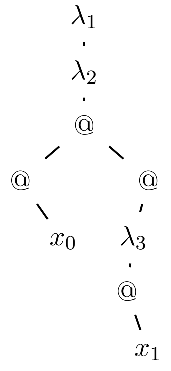
The syntax tree of our example term in de Bruijn notation is



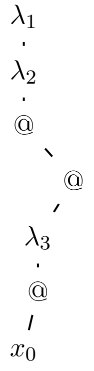
In this case the bound variable b has two instances which are denoted by x_0 and x_1, while x_0 (resp. x_1) can denote both c or b (resp. b or a). Since bound variables are indexed as described in Section 3.3, this gives a somewhat scattered information on where to find the variables that are bound by one particular λ-binder. This kind of information, however, is important in practice, for example, to support efficient β-reduction. We therefore once more employ PSTs to describe the occurrences of variables bound by one and the same λ-binder:



Variables bound by λ_1



Variables bound by λ_2



Variables bound by λ_3

For example, the PST indicating occurrences of variables bound by λ_2 in the above example is given by:

$$
\begin{array}{ll}
p_1 = pst(p_2, \_, \_) & \\
p_2 = pst(p_3, \_, \_) & \\
p_3 = pst(\_, p_4, p_6) & \\
p_4 = pst(\_, \_, p_5) & \\
p_5 = pst(\_, \_, \_) & \text{with annotation 0} \\
p_6 = pst(\_, p_7, \_) & \\
p_7 = pst(p_8, \_, \_) & \\
p_8 = pst(\_, \_, p_9) & \\
p_9 = pst(\_, \_, \_) & \text{with annotation 1}
\end{array}
$$

The explicit notation of variables bound by λ_2 and λ_3 is analogous and therefore omitted here.

This list of PSTs is also recorded in LEO-II's index, in the order shown above. Each PST is assigned a *scope number*, where the scopes are defined by the occurrence of λ-binders. When traversing the syntax tree, the PST recording occurrences of variables bound by the *first* λ-binder is assigned scope number 1, the PST related to the *second* λ-binder has scope number 2 and so on.

While the term $\lambda((\lambda.x_0) = ((\lambda.(x_0 x_1))x_1))$ is closed, that is, all de Bruijn indexed variables are bound by a λ-binder within the term, this is not always true for its subterms. Unbound variables occur, for example, in $\lambda.(x_0 x_1)$, where x_1 refers to a binder outside the term. In particular, all primitive terms consisting only of de Bruijn variables refer to a binder outside this term. While the PSTs for bound variables can be constructed as shown above, the determination of the scope numbers deserves a special treatment in case of *loose bound variables*, that is bound variables without a binder in the given subterm. If a term has occurrences of loose bound variables, their de Bruijn index allows to determine the distance to their (virtual) binder measured in scopes upwards from the term's root position. PSTs for loose bound variables are assigned a scope number $s \leq 0$. The PST to denote all occurrences of x_1 in itself is consequently assigned the scope number -1, and the PST denoting the occurrence of x_1 in $\lambda.(x_0 x_1)$ is assigned scope number 0.

Indexing of bound variable occurrences in a term is used to speedup β-reduction. For each λ-binder the positions of variables bound by this binder are known, thus, only the parts of the term that actually are modified have to be processed. In the context of explicit substitutions [FKP96, ACCL90], the implementation of shift and lift operators can furthermore be reduced to recalculation of the offset and elimination of bound variable PSTs from the list.

3.5 Using the Index

3.5.1 Adding and Retrieving Terms:

Terms are added to and retrieved from an index in a similar way as in coordinate or path indexing [Sti89]. When a term t is added to the index, the PSTs of symbol occurrences are constructed as described in Section 3.3. Then the following hashtables are updated:

- In `occurrences`, the PSTs of the occurring symbols (or subterms) are added.
- For each occurring symbol (or subterm), t is added to the set of terms which is recorded for that symbol in `occurs_in`. If there is no such entry in `occurs_in`, the singleton $\{t\}$ is added.
- For each term position in t, t is indexed in `occurs_at` in the same way with the position as first key and the the subterm as second key.

Furthermore, the PSTs for bound variables are constructed as described in Section 3.4 and are added to the hashtable `boundvars`.

For each occurrence of a subterm at a given position in a query term, a set of candidate terms is retrieved from hashtable `occurs_at`. Sets of candidate terms can furthermore be retrieved from hashtable `occurrences` for subterms occurring at unspecified positions. The result of the query is the intersection of all candidate sets obtained this way.

3.5.2 Speeding up Computation:

Using the index, efficient occurs checks are reduced to single hashtable lookups. Efficient replacement of a symbol or subterm t is furthermore supported by PSTs recorded in the index (in hashtable `occurrences`), since these PSTs make it possible to avoid processing of term parts with no occurrences of t. Fast β-reduction is supported by the PSTs recorded in hashtable `boundvars`.

3.5.3 Explicit Substitutions:

Our approach can also support explicit substitutions. Note that subterm occurrences in a term t can be quickly determined as described above. Similarly, the occurrences of a subterm s in the result of applying a substitution σ to a term t can be determined using our indexing technique. To determine occurrences of s in σt where $\sigma = [b/a]$, occurrences of s and a in t are looked up from the index, as well as occurrences of s in b. Thus we get three PSTs $pst_{s/t}$, $pst_{a/t}$ and $pst_{s/b}$. To find all occurrences of s in σt, all positions annotated by a in $pst_{a/t}$ are replaced by a new sub PST $pst_{s/b}$, and the result is merged with $pst_{s/t}$. For $\sigma_1\sigma_2 \ldots \sigma_n t$, this operation is cascaded.

4 Preliminary Evaluation

Full evaluation of the indexing method presented here is still work in progress. This is because the implementation of LEO-II is still at its very beginning, so that we cannot pursue an empirical evaluation within theorem proving applications with LEO-II at the current stage of development. A purely theoretical examination is difficult and furthermore questionable, as the computational complexity can be expected to heavily depend on the structure of the application domains [NHRV01].

However, we were able to undertake some first experiments which may give us an impression of the efficiency gain we may expect for LEO-II (for example, in comparison to LEO and other higher order theorem provers that do not use term indexing techniques).

In order to get a realistic impression of the structural characteristics of real world term sets, we indexed a sample selection of 900 theorems and definitions from a HOTPTP [GS06] version of Jutting's Automath encoding of Landau's book Grundlagen der Analysis [vBJ77, Lan30]. An overview on the results of this experiment is given in Figure 1. We will now discuss these results.

In our study, we determined, for example, the rate of term sharing, which is the average number of parent nodes per node and the average number of terms a given node occurs in. At first sight the average number of parent nodes of 1.68 appears to be relatively low, an impression which is underlined by the high number of nodes with no or one parent node (about 90%). For nodes which are deeply buried in a term's structure, however, the sharing rate multiplies along the path up to root position, so the average number of terms a node occurs in (33.5) relativises this impression. Our experiments indicate furthermore an increase of the sharing rate by operations like

Number of indexed terms	977
Number of created term nodes	11618
Average term size	54
Number of nodes with no parent nodes	904
Number of nodes with one parent node	9633
Number of nodes with two more more parent nodes	1083
Maximum number of parent nodes	2778 (symbol $\forall$)
Average number of parent nodes	1.68
Average number of terms a node occurs in	33.5
-"-(for symbols)	493.9
-"-(for nonprimitive term nodes)	24
Average PST/term size for symbol occurrences	0.21
Average PST/term size for bound variable occurrences	0.33
Average PST/term size for all term nodes	0.12

Figure 1: Structure of the Landau sample.

replacement, substitution and β-reduction, due to the reuse of already indexed subterms. Additionally, the maintenance of the index is supported by data already existing in the index. As most logical operations on terms reuse parts of these terms, the cost to maintain the index is less than indexing a set of terms starting from an empty index, as required for instance, when initially loading a mathematical theory to memory.

An indicator for the term retrieval performance is the average number of terms a node occurs in. With an average number of occurrences of 33.5 and a total of 11618 term nodes, a theoretical average of 99.7% of candidate nodes for retrieval can be excluded by checking occurrences of subterms only (compared to a naive approach). By specifying the position of the subterm's occurrence, the set of retrieved terms is further restricted.

The use of shared terms is responsible for a further improvement of performance similar to the transition from coordinate indexing to path indexing [Sti89]. While both methods employ a common underlying idea, path indexing is substantially faster. In the former approach terms are discriminated by occurrences of single symbols at specified positions (or *coordinates*, hence the name). The criterion in the latter is the occurrence of a *path*, that is the occurrence of a sequence of specified symbols in a descending path in the syntax tree. The retrieval of candidates for one path of length n is thus corresponding to n passes of retrieval in coordinate indexing. We expect a similar effect in our approach due to shared representation of terms, since terms are indexed according to the occurrence of nonprimitive term structures, too. This assumption is supported by the increase of the exclusion rate of 95.7% for symbol occurrences only (with an average number of 493.9 superterms per node) to 99.8% for nonprimitive terms (with an average of 24 superterms per node). This rise corresponds to a theoretical speedup by factor 20.

We can also predict a significant performance improvement of operations, such as replacement, substitution and occurs check. They all are critical in theorem proving. The indexing method we present here supports occurs checks in constant time, based on simple hashtable lookup. This applies not only to symbols, but also to nonprimitive terms. This also supports global replacement of defined terms (for example, $a = b$) by their definiendum (for example, $\forall P.Pa \Rightarrow Pb$).

A measure of the efficiency improvement for replacement operations is the PST/term size rate. The value is 0.21 for symbols, which is relevant, for example, for variable substitution, and which corresponds to a theoretical speedup by factor 5. The value for bound variables, which is relevant for β-reduction, is 0.33, corresponding to a theoretical speedup by factor 3. The probably least common operation is the replacement of nonprimitive terms, as discussed above.

We are aware of the fact that the results shown here are based on a theoretical juggling with average values. These results may thus differ strongly from the behaviour when used in a realistic application in theorem proving as we intend. This is due to several factors: First we expect the structure of the set of indexed terms to change during operation. In general the basic operations of a theorem prover will increase the sharing rate of some symbols and subterms. This makes occurrences of these terms a less discriminating criterion, on the other hands it decreases the cost of maintenance of the index. The tradeoff of these two factors will be subject to further examination. Second, the evaluation of average values does most likely not correspond to index operation sequences as they actually occur in a realistic theorem proving application.

5 Conclusion and Future Work

The main features of the new higher order term indexing method we presented in this paper are shared term representation, relational indexing of subterm occurrences and the use of partial syntax trees. Occurrences of subterms are indexed in several ways and can be flexibly combined to design customised procedures for term retrieval and heuristics. Our method furthermore provides support for potentially costly operations such as global unfolding of defined terms. Our indexing method is based on simple hashtable operations, so there is little computational overhead in term retrieval and maintenance of the index. Indexing of subterm occurrences allows furthermore for an occurs check in constant time. Additionally, the performance is improved by the use of PSTs. Finally, a shared representation of terms helps to keep the costs for maintaining the index low and improves the performance of retrieval operations.

The indexing technique presented in this paper has been implemented in OCaml [LDG$^+$05]. A proper evaluation of the approach within a real theorem proving context is still work in progress. However, first experiments are promising.

The preliminary evaluation in this paper is based on some statistical data we computed for 900 example terms from an encoding of Landau's textbook. To what extend our predictions on efficiency gain are realistic will be examined in future work.

Furthermore, as the experience from first order term indexing shows, most successful systems employ a combination of various indexing methods which are used complementarily. We will thus also evaluate which aspects of our indexing method result in a real performance gain and which do not. Our evaluation will be done with the LEO-II prover as soon as a first version of its resolution loop is available. In the LEO-II context we are particularly interested in the fast determination of clauses (resp. literals and terms in clauses) with respect to certain filter criteria. In the extreme case, these criteria may be based on complex operations such as higher order pattern unification [PP03] or even full higher order unification [Hue75, SG89].

Our approach differs from Pientka's work, which has a stronger emphasis on term retrieval. Pientka's method is based on high level operations such as unification of linear higher order patterns to construct substitution trees, while our method relies mainly on simpler low level operations and makes strong use of hashtables. Both methods appear complementary to some extend, which motivates the study of a combination of both.

Future work also includes the investigation of alternative term representation techniques, such as suspension calculus [Nad02], spine representation [CP97] and explicit substitutions [FKP96, ACCL90] in the context of our term indexing approach. We are especially interested in the combination of aspects from different representation techniques within a single graph structure.

References

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[Ame92] American National Standards Institute. *American national standard for information systems: database language — SQL: ANSI X3.135-1992*. American National Standards Institute, pub-ANSI:adr, 1992. Revision and consolidation of ANSI X3.135-1989 and ANSI X3.168-1989, Approved October 3, 1989.

[BK98] Christoph Benzmüller and Michael Kohlhase. LEO – a higher-order theorem prover. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, number 1421 in LNAI, pages 139–143, Lindau, Germany, 1998. Springer.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CP97] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Pittsburgh, PA, 1997.

[dB72] N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[FKP96] Maria C. F. Ferreira, Delia Kesner, and Laurence Puel. Lambda-calculi with explicit substitutions and composition which preserve beta-strong normalization. In *Algebraic and Logic Programming*, pages 284–298, 1996.

[GS06] Allen Van Gelder and Geoff Sutcliffe. Extending the tptp language to higher-order logic with automated parser generation. In *Proceedings of IJCAR*, 2006.

[Hue75] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Lan30] Edmund Yehezkel Landau. *Grundlagen der Analysis.* Erstveröff, Leipzig, first edition, 1930.

[LDG$^+$05] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Remy, and Jerome Vouillon. *The Objective Caml system, release 3.09*, October 2005.

[McC92] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[Nad02] G. Nadathur. The suspension notation for lambda terms and its use in metalanguage implementations, 2002.

[NHRV01] R. Nieuwenhuis, T. Hillenbrand, A. Riazanov, and A. Voronkov. On the evaluation of indexing techniques for theorem proving. In *Proceedings of IJCAR 2001*, volume 2083 of *LNAI*, pages 257–271. Springer Verlag, 2001.

[Pie03] Brigitte Pientka. Higher-order substitution tree indexing. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2003.

[PP03] B. Pientka and F. Pfenning. Optimizing higher-order pattern unification, 2003.

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AICOM*, 15(2-3):91–110, jun 2002.

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[SG89] W. Snyder and J. Gallier. Higherorder unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(2):101–114, 1989.

[Sti89] M. Stickel. The path-indexing method for indexing terms. Technical Report 473, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, 1989.

[TSP06] Frank Theiß, Volker Sorge, and Martin Pollet. Interfacing to computer algebra via term indexing. In *Proceedings of Calculemus.* Elsevier, 2006.

[vBJ77] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system.* PhD thesis, Technische Hogeschool Eindhoven, Stichting Mathematisch Centrum, 1977. See also: [Lan30].

[WBH$^+$02] Ch. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, Ch. Theobald, and D. Topic. SPASS version 2.0. In *Proceedings of CADE 2002*, pages 275–279, 2002.

Integrating External Deduction Tools with ACL2 [*]

Matt Kaufmann, J Strother Moore, Sandip Ray, Erik Reeber
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
{kaufmann,moore,sandip,reeber}@cs.utexas.edu
http://www.cs.utexas.edu/users/{kaufmann,moore,sandip,reeber}

Abstract

We present an interface connecting the ACL2 theorem prover with external deduction tools. The logic of ACL2 contains several constructs intended to facilitate structuring of interactive proof development, which complicates the design of such an interface. We discuss some of these complexities and develop a precise specification of the requirements from external tools for sound connection with ACL2. We also develop constructs within ACL2 to enable the developers of external tools to satisfy our specifications.

1 Introduction

Recent years have seen rapid advancement in the capacity of automatic reasoning tools, most notably for decidable theories such as propositional calculus and Presburger arithmetic. For instance, modern BDD packages and Boolean satisfiability solvers can automatically solve problems with tens of thousands of variables and have been successfully used to reason about commercial hardware system implementations. This advancement has sparked significant interest in the general-purpose, interactive theorem proving community to improve the efficiency and automation in theorem provers by providing a connection with state-of-the-art automatic reasoning tools. In this paper, we present a general mechanism we are building to connect deduction tools, external to the ACL2 theorem prover, with that prover.

ACL2 [KMM00, KM06] is an industrial-strength interactive theorem proving system. It consists of an efficient programming interface based on an applicative subset of Common Lisp [KM94], and a first-order, inductive theorem prover for a logic of recursive functions. The ACL2 theorem prover supports several deduction mechanisms such as congruence-based conditional rewriting, well-founded induction, several decision procedures, and generalization. The theorem prover has been particularly successful in the verification of microprocessors and hardware designs such as

[*]This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591, by the National Science Foundation under Grant No. ISS-0417413, and by DARPA under Contract No. NBCH30390004.

the floating point multiplication, division, and square root algorithms of AMD processors [MLK98, Rus98, RF00, FKR$^+$02], microcode for the Motorola CAP DSP [BH97], separation properties for the Rockwell Collins AAMP7$^{\mathrm{TM}}$ processor [GRW04], and a non-trivial pipelined machine with interrupts, exceptions, and speculative instruction execution [SH97]. However, the applicability of ACL2 (as in fact that of any theorem prover) is often limited by the amount of user expertise required to drive the theorem prover; indeed, the verification projects referenced above represent many man-years of effort. Yet, a significant number of lemmas proven in the process, in particular many proofs exhibiting invariance of predicates over executions of hardware design implementations, can be expressed in a decidable theory and automatically dispatched by an automatic decision procedure for the theory.

On the other hand, it is not trivial to connect ACL2 with an external deduction tool. The logic of ACL2 is complicated by the presence of several constructs intended to facilitate effective proof structuring [KM01]. It is therefore imperative (i) to determine under what logical constraints a conjecture certified by a combination of the theorem prover and other tools can be claimed to be a valid theorem, and (ii) to provide mechanisms so that a tool implementor might be able to meet the logical constraints so determined.

In this paper, we propose a general interface for connecting external tools with ACL2. The user can instruct ACL2 to use external deduction tools for reducing a goal formula C to a list of formulas L_C during a proof attempt. The claim is that provability of each formula in L_C implies the provability of C. We present a sufficient condition expressible in ACL2 guaranteeing this claim, and discuss the soundness requirements on the tool implementor. We also propose a modest augmentation of the logical guarantees provided by ACL2, in order to facilitate connection with certain types of tools (cf. Section 5).

We distinguish between two classes of external tools, namely (i) tools verified by the ACL2 theorem prover, and (ii) unverified but trusted tools. A verified tool must be formalized in the logic of ACL2 and the sufficient condition alluded to above must be formally established by the theorem prover. An unverified tool can be defined using the ACL2 programming interface, and can invoke arbitrary executable programs using calls to the underlying operating system via a system call interface. An unverified tool is introduced with a "tag" acknowledging that the validity of the formulas proven using the tool depends on the correctness of the tool.

The interface for unverified tools enables us to invoke Boolean Satisfiability solvers, BDD packages, etc., for simplifying ACL2 subgoals. Why might *verified* tools be of interest? The formal language of ACL2 is a programming language, based on an applicative subset of Common Lisp. The close connection with Lisp makes it possible to write efficiently executable programs in the ACL2 logic [KM94]. In fact, most of the ACL2 source code is implemented in this language. We believe it will be handy to provide facilities to the ACL2 user to control proofs by (i) implementing customized domain-specific reasoning code, (ii) verifying with ACL2 that the code is sound, and (iii) invoking the code for proving theorems in the target domain. In fact, ACL2 currently provides a way for users to augment its built-in term simplifier with their own customized reasoning code, via the so-called "meta rules" [BM81]. However, such rules essentially augment the reasoning engine of ACL2 without providing the user control to manipulate a specific subgoal arising during a proof. Furthermore, meta rules only allow reducing a term to one that is provably equivalent—they do not allow generalization.

With our interface, an ACL2 user can invoke directly a customized, verified reasoning tool to replace a subgoal by a collection of possibly more general subgoals.

The remainder of the paper is organized as follows. In Section 2 we provide a brief overview of the ACL2 system. In Sections 3 through 5 we present our interface for connecting verified and unverified external tools with ACL2, touching upon the logical underpinnings involved. We discuss related work in Section 6 and conclude in Section 7. No previous familiarity with ACL2 is assumed in this presentation; the relevant features of the logic and the theorem prover are discussed in Section 2.

2 ACL2

The name "ACL2" stands for "A Computational Logic for Applicative Common Lisp". The name is used to denote (i) a programming language based on an applicative subset of Common Lisp, (ii) a first-order logic of recursive functions with induction, and (iii) a theorem prover for the logic. In this section, we provide a brief overview of ACL2. The review is not complete, but only intended to lay the foundation for our work. Readers interested in learning ACL2 are referred to the ACL2 Home Page [KM06] which contains extensive hypertext documentation together with references to several books and papers.

2.1 The logic

The kernel of the ACL2 logic consists of a formal syntax, some rules of inference, and some axioms. Kaufmann and Moore [KM97] provide a precise description of the kernel logic. The logic supported by the theorem prover is an extension of the kernel logic.

The kernel syntax describes terms composed of variables, constants, and function symbols applied to a fixed number of argument terms. The kernel logic introduces "formulas" as composed of equalities between terms and the usual propositional connectives.

The syntax of ACL2 is the prefix-normal syntax of Lisp; thus, the application of a binary function f on arguments a and b is represented by `(`f a b`)` rather than the more traditional $f(a, b)$. However, in this paper we will use the formal syntax only when it is relevant for the associated discussion. In particular we will write $(x \times y)$ instead of `(* x y)` and (*if* x *then* y *else* z) instead of `(if x y z)`.

The axioms of ACL2 describe the properties of certain Common Lisp primitives. For example, the following are axioms about the primitives *equal* and *if*:

Axioms.

$$\begin{array}{lll} x = y & \Rightarrow & \mathit{equal}(x,y) = \mathtt{T} \\ x \neq y & \Rightarrow & \mathit{equal}(x,y) = \mathtt{NIL} \\ x = \mathtt{NIL} & \Rightarrow & (\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z) = z \\ x \neq \mathtt{NIL} & \Rightarrow & (\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z) = y \end{array}$$

Notice that the kernel syntax is quantifier-free and each formula is implicitly universally quantified over all the free variables in the formula. Furthermore, the use of function symbols *equal* and *if* make it possible to embed propositional calculus and equality into the term language. When we write a term τ in place of a formula, it stands for the formula $\tau \neq \mathtt{NIL}$. Thus, in ACL2, the following term is an axiom relating the Lisp functions *cons*, *car*, and *equal*.

Axiom.
$\mathit{equal}(\mathit{cons}(\mathit{car}(x, y)), x)$

This axiom stands for the formula $\mathit{equal}(\mathit{car}(\mathit{cons}(x, y)), x) \neq \texttt{NIL}$, which is provably equivalent to $\mathit{car}(\mathit{cons}(x, y)) = x$. With this convention, we will feel free to interchange terms and formulas. We will similarly feel free to apply logical connectives to a term or formula. Thus when we write $\neg\tau$, where τ is a term, we mean the term (or formula by the above convention) obtained by applying the function symbol *not* to τ, where *not* is axiomatized as:

Axiom.
$\mathit{not}(x) \ = \ \mathit{if}\ x\ \ \mathit{then}\ \texttt{NIL}\ \ \mathit{else}\ \texttt{T}$

The convention above enables us to interpret an ACL2 theorem as follows. If the term τ (when interpreted as a formula) is a theorem then for all substitutions σ of free variables in τ to objects in the ACL2 universe the (ground) term τ/σ evaluates to a non-`NIL` value. This alternative view will be critical in deriving sufficient conditions for correctness of external tools integrated with ACL2.

The kernel logic includes axioms that characterize the primitive Lisp functions over numbers, characters, strings, symbols, and ordered pairs. These objects together make up the ACL2 standard universe; but "non-standard" ACL2 universes may contain other objects. Lists are represented as ordered pairs, so that the list `(1 2 3)` is represented as $\mathit{cons}(1, \mathit{cons}(2, \mathit{cons}(3, \texttt{NIL})))$. For brevity, we will write $\mathit{list}(x, y, z)$ as an abbreviation for $\mathit{cons}(x, \mathit{cons}(y, \mathit{cons}(z, \texttt{NIL})))$. Another convenient data structure built out of ordered pairs is the *association list* (or *alist*) which is essentially a list of pairs, *e.g.*, $\mathit{list}(\mathit{cons}(\texttt{"a"}, 1), \mathit{cons}(\texttt{"b"}, 2))$. We often use alists for describing finite mappings; the above alist can be thought as a mapping that associates the strings `"a"` and `"b"` with 1 and 2, respectively.

In addition to propositional calculus and equality the rules of inference of ACL2 include instantiation, together with first-order induction over ϵ_0 (see below). For instance, the formula $\mathit{car}(\mathit{cons}(2, x)) = 2$ is provable by instantiation from the above axiom relating *car*, *cons*, and *equal*.

The ACL2 theorem prover initializes with a boot-strapping (first-order) theory called the *Ground Zero* theory (GZ for short). In the sequel, whenever we mention an ACL2 theory, we mean a theory obtained by extending GZ via the *extension principles* explained below. The theory GZ contains the axioms of the kernel logic. In addition, it also contains a well-founded first-order induction principle, by way of an embedding of ordinals below ϵ_0. In particular, GZ is assumed to be *inductively complete*, that is, it is assumed implicitly to contain all the first-order well-founded induction axioms expressible using formulas ϕ in the language of GZ:

$$(\forall y < \epsilon_0)[((\forall x < y)\phi/\{y := x\}) \Rightarrow \phi(y)] \Rightarrow (\forall y < \epsilon_0)\phi(y)$$

2.1.1 Extension Principles

ACL2 also provides several *extension principles* that allow the user to extend a theory by introducing new function symbols and axioms about them. Two extension principles that are particularly relevant to us are (i) the *definitional principle* to introduce total functions, and (ii) the *encapsulation principle* to introduce constrained functions,[1] and we discuss them in some detail. Note that whenever we say (below) that a theory is extended by axiomatizing new function symbols we implicitly assume that the resulting theory is also inductively complete, that is, all the induction axioms in the language of the extended theory are also introduced together with the axioms explicitly specified.

Definitional Principle:

The *definitional principle* allows the user to extend a theory by axiomatizing new total (recursive) functions. For example, one can use this principle to introduce the unary function symbol *fact* axiomatized as follows, which returns the factorial of its argument.

Definitional Axiom.
$\mathit{fact}(n)\ =\ \mathit{if}\ \mathit{natp}(n) \wedge (n > 0)\ \ \mathit{then}\ \ n \times \mathit{fact}(n-1)\ \ \mathit{else}\ 1$

Here, $\mathit{natp}(n)$ is axiomatized in GZ to return T if n is a natural number, and NIL otherwise. To ensure that the extended theory is consistent, ACL2 first proves that the recursion terminates. This is achieved by exhibiting some *measure* m that maps the set of function arguments to some well-founded structure derived from the embedding of ordinals below ϵ_0. For the axiom above, an appropriate measure is $\mathit{nfix}(n)$ which is axiomatized in GZ to return n if n is a natural number, otherwise 0.

Encapsulation Principle:

The *encapsulation principle* allows the extension of the ACL2 logic with functions introduced with constraints rather than full definitions. This principle, for instance, allows us to extend a theory by introducing a new unary function *foo* with only the following axiom that merely posits that *foo* always returns a natural number:

Encapsulation Axiom.
natp(*foo*(x))

The encapsulation axioms are also referred to as *constraints*, and the functions introduced via this principle are called *constrained functions*. To ensure the consistency of the resulting theory, one must show that there exist (total) functions satisfying the alleged constraints; such functions are called *witnesses* to the constraints. For *foo* above, an appropriate witness is the constant function that always returns 1.

For a constrained function f the only axioms known are the constraints. Therefore, any theorem proved about f is also valid for a function f' that also satisfies the constraints. More precisely, call the conjunction of the constraints on f the formula ϕ. For any formula θ, let θ' be the formula obtained by replacing the function symbol f by the function symbol f'. Then a derived rule of inference, *functional instantiation*, specifies that if ϕ' and ψ are theorems then ψ' is also a theorem. Consider, for example, the constant function of one argument that returns 10. This function satisfies the constraint for *foo*; thus if $\mathit{bar}(\mathit{foo}(x))$ is provable for some function *bar* then functional instantiation can be used to prove $\mathit{bar}(10)$. .

[1]Other extension principles include the introduction of Skolem (choice) functions and specification of a formula as an axiom. The latter is discouraged since one can introduce unsoundness by adding arbitrary axioms. For this paper, we will ignore the possibility of introducing arbitrary axioms.

2.2 The Theorem Prover

As a theorem prover, ACL2 is an automated, interactive proof assistant. It is *automated* in the sense that no user input is expected once the theorem prover has embarked on the search for the proof of a conjecture. It is *interactive* in the sense that the proof search is largely determined by the previously proven lemmas in its database at the beginning of a proof attempt; the user essentially programs the theorem prover by stating lemmas for it to prove, to use automatically in subsequent proofs. There is also a goal-directed interactive loop (called the "*proof-checker*"), similar in nature to what is offered by LCF-style provers; but it is much less frequently used and not relevant to the discussion below.

Interacting with the ACL2 theorem prover principally proceeds as follows. The user creates a relevant theory (extending GZ) using the extension principles to model some artifact of interest. Then she poses some conjecture about the functions in the theory and instructs the theorem prover to prove the conjecture, possibly providing hints on how to proceed in the proof search. For instance, if the artifact is the factorial function above, an appropriate conjecture might be the following formula, which says that *fact* always returns a natural number.

Theorem `fact-is-natp`:
  natp(*fact*(x)) = T

The theorem prover attempts to prove such a conjecture by applying a sequence of transformations to it, replacing each goal (initially, the conjecture) with a list of subgoals. ACL2 provides a *hint* mechanism that enables the user to instruct the theorem prover on how to proceed with its proof search at any goal or subgoal. For instance, the user can instruct the theorem prover to begin its proof search by inducting on `x`.

Once a theorem is proven, the theorem prover stores it in a database, for use in subsequent derivations. This database groups theorems into various *rule classes*, which affects how the theorem prover will automatically apply them. The default rule class is `rewrite`, which causes the theorem prover to replace instances of the left-hand-side of an equality with its corresponding right-hand-side. If the conjecture `fact-is-natp` above is a rewrite rule, then subsequently whenever ACL2 encounters a term of the form *natp*(*fact*(τ)) in the course of a proof attempt, it rewrites the term to T.

ACL2 users interact with the theorem prover primarily by issuing a sequence of *event* commands for introducing new functions and proving theorems with appropriate rule classes. For example, `fact-is-natp` is the name of the above theorem event. During proof development the user typically records events in a file, often referred to as a *book*. Once the desired theorems have been proven, the user instructs ACL2 to *certify* such a book in order to facilitate the use of the events in other projects. A book can be certified once and then *included* during a subsequent ACL2 session without rerunning the associated proofs. To facilitate structured proof development, the user is permitted to mark some of the events in a book as *local events*. For instance, to prove some relevant theorem the user might introduce several auxiliary functions and intermediate lemmas that are not generally useful; such events are typically marked to be local. When a book is included in a subsequent proof project, only the non-local events in the book are accessible, thus preventing unwanted clutter in the database of the theorem prover.

The presence of local events complicates the soundness claims for ACL2. Note from above that local events in a book might include commands for introducing new functions (thus extending an ACL2 theory with new axioms), which are not available in the subsequent sessions where the book is loaded. Yet, in order to prove some non-local theorem in the book ACL2 might have used some of these local axioms. One must therefore answer under what condition it is legitimate to mark an axiomatic event in a book as local, and what formal soundness claims can be provided for an ACL2 session in which such a pre-certified book is loaded. Such questions have been answered by Kaufmann and Moore [KM01]: if a formula ϕ is proven as a theorem in an ACL2 session, then ϕ is in fact first-order derivable (with induction) from the axioms of GZ together with (hereditarily) only the axiomatic events in the session that involve the function symbols in ϕ. (In particular, every ACL2 session corresponds to a theory that is a *conservative* extension of GZ.) Thus, any definition or theorem that does not involve the function symbols in the non-local events of a book can be marked local. To implement this requirement, book certification involves two passes. In the first pass, ACL2 proves each theorem (and admits each axiomatic event) sequentially. In the second pass, it skips proofs, and makes a so-called *local incompatibility* check, checking primarily that each axiomatic event involved in any non-local theorem in the book is also non-local.

2.3 The ACL2 Programming Environment

ACL2 is closely tied with Common Lisp. The formal syntax of the logic is essentially the syntax of Lisp, and the axioms in GZ for the primitive Lisp functions are carefully crafted so that the return value of a function as predicted by the axioms matches with the value specified in the Common Lisp Manual on arguments in the intended domain of its application. Furthermore, events corresponding to functions introduced using the definitional principle are essentially Lisp definitions. For instance, consider the factorial function *fact* described above. The formal event introducing the definitional axiom of *fact* is written in ACL2 as follows.

```
(defun fact (n) (if (and (natp n) (> n 0)) (* n (fact (- n 1))) 1))
```

This is essentially a Lisp definition of the function! The connection with Common Lisp enables the users of ACL2 to execute formal definitions by using the underlying Lisp evaluator. Since Lisp is an ANSI-standard, efficient functional programming language, ACL2 users often make use of the connection to implement formally defined yet efficient code. Indeed, the theorem prover itself makes use of this connection for simplifying ground terms during proof search; for instance, ACL2 will simplify *fact*(3) to 6 by evaluation in the underlying Lisp.

In order to facilitate efficient code development, ACL2 also provides a logic-free programming environment. A user can implement any applicative Lisp function and mark it to be in *program mode*. No proof obligation is generated for such functions. ACL2 can evaluate such functions using the Lisp evaluator, although no logical guarantee (including termination) is provided. Furthermore, ACL2 provides an interface to the underlying operating system, which enables the user to invoke arbitrary executable code (and operating system commands) from inside an ACL2 session.

2.4 Evaluators

ACL2 provides a convenient notation for defining an evaluator for a fixed set of functions. Evaluators are used to support *meta reasoning* [BM81]. We will not consider meta reasoning in this paper, but we briefly mention evaluators since they will be useful in characterizing the correctness of external tools.

A proof search involves applying transformations to reduce a goal to a collection of subgoals. Internally, ACL2 stores each goal as a *clause* represented as an object in the ACL2 universe. For instance, when ACL2 attempts to prove a theorem of the form $\tau_1 \wedge \tau_2 \wedge \ldots \wedge \tau_n \Rightarrow \tau$, it represents the proof goal internally as a list of terms, $(\neg\tau_1 \ \ldots \ \neg\tau_n \ \tau)$, which can be thought of as the disjunction of its elements (*literals*). When ACL2 works on any subgoal, the transformation procedures work on the internal representation of the subgoal, called the *current clause*. Since this representation is an ACL2 object, we can define functions over such objects.

An *evaluator* makes explicit the connection between terms and their internal representations. Assume that $f_1, \ldots, f_n$ are functions axiomatized in some ACL2 theory $\mathcal{T}$. A function ev, also axiomatized in $\mathcal{T}$ is called an *evaluator* for $f_1, \ldots, f_n$, if the axioms associated with ev can be viewed as specifying an evaluation semantics for the internal representation of terms composed of $f_1, \ldots, f_n$ that is consistent with the definitions of these functions; such axioms are then referred to as *evaluator axioms*. A precise characterization of all the evaluator axioms is described in the ACL2 Manual [KM06] under the documentation topic defevaluator; here we only mention one for illustration, which corresponds to the evaluation of the m-ary function symbol f_i:

An Evaluator Axiom.
$\mathit{ev}(\mathit{list}('\mathtt{f_i}, '\tau_1, \ldots, '\tau_\mathtt{m}), a) = f_i(\mathit{ev}('\tau_1, a), \ldots, \mathit{ev}('\tau_m, a))$

Here $'f_i$ is assumed to be the internal representation of f_i and $'\tau_j$ is the internal representation of τ_j, for $1 \leq j \leq m$. It is convenient to think of a as an association list that maps the (internal representation of the) free variables in $\tau_1, \ldots, \tau_m$ to ACL2 objects. Then the axiom specifies that the evaluation of the list $('f_i \ '\tau_1 \ \ldots \ '\tau_m)$ (which corresponds to the internal representation of $f_i(\tau_1, \ldots, \tau_m)$) under some mapping of free variables to objects is the same as the function f_i applied to the evaluation of each τ_j under the same mapping.

3 Verified External Tools

In this section, we discuss *verified* external tools. We consider verified tools first since they are amenable to perhaps a simpler understanding than unverified ones. The ideas and infrastructure we develop in this section will be extended successively in the next two sections to support connections with unverified tools.

We will refer to external deduction tools as *clause processors*. Recall that ACL2 internally represents terms as clauses, so that a subgoal of the form $\tau_0 \wedge \tau_1 \wedge \ldots \wedge \tau_n \Rightarrow \tau$ is represented as a disjunction by the list $(\neg\tau_0 \ \neg\tau_1 \ \ldots \ \neg\tau_n \ \tau)$. Our interface enables the user to transform the current clause with custom code. More precisely, a *clause processor* is a function that takes a clause C (together with possibly other arguments)

$$\begin{array}{lcl} \textsf{\textit{disjoin}}(C) & = & \textsf{\textit{if}}\ \neg\textsf{\textit{consp}}(C)\ \textsf{\textit{then}}\ *\texttt{NIL}*\ \textsf{\textit{else}}\ \textsf{\textit{list}}(\texttt{if}, \textsf{\textit{car}}(C), *\texttt{T}*, \textsf{\textit{disjoin}}(\textsf{\textit{cdr}}(C))) \\ \textsf{\textit{conjoin}}(L_C) & = & \textsf{\textit{if}}\ \neg\textsf{\textit{consp}}(L_C)\ \textsf{\textit{then}}\ *\texttt{T}*\ \textsf{\textit{else}}\ \textsf{\textit{list}}(\texttt{if}, \textsf{\textit{disjoin}}(\textsf{\textit{car}}(L_C)), \textsf{\textit{conjoin}}(\textsf{\textit{cdr}}(L_C)), *\texttt{NIL}*) \end{array}$$

Figure 1: Axioms to support clause processors in GZ. Here `*T*` and `*NIL*` are assumed to be the internal representation of `T` and `NIL` respectively. The predicate *consp* is defined in GZ such that *consp*(x) returns `T` if x is an ordered pair, and `NIL` otherwise.

and returns a list of clauses L_C.[2] The intention is that if each clause in L_C is a theorem of the current ACL2 theory then so is C. In the remainder of the paper, when we talk about clause processors, we will mean such clause manipulation functions.

Our interface for verified external tools constitutes the following components.

- *A new rule class for installing clause processors.* Suppose the user has defined a function *tool0* that she desires to use as a clause processor. She can then prove a specific theorem about *tool0* (described below) and attach this rule class to the theorem. The effect is to install *tool0* in the ACL2 database as a clause processor for use in subsequent proof attempts.

- *A new hint for using clause processors.* Once *tool0* has been installed as a clause processor it can be invoked via this hint to transform a conjecture during a subsequent proof attempt. If the user instructs ACL2 to use *tool0* to help prove some goal G, then ACL2 transforms G into the collection of subgoals generated by executing *tool0* on (the clause representation of) G.

We now explain the theorem alluded to above for installing a function *tool0* as a clause processor. Recall that one way to interpret a formula proven by ACL2 is via an evaluation semantics; that is, a formula Φ is a theorem if, for every substitution σ mapping each free variable of Φ to some object, Φ/σ does not evaluate to `NIL`. Our formal proof obligation for installing functions as clause processors is based on this evaluation semantics. Let C be a clause whose disjunction is the term τ, and let *tool0*, with C as its argument, produce the list (C_1 ... C_n) whose respective disjunctions are the terms $\tau_1, \ldots, \tau_n$. Informally, we want to ensure that if τ/σ evaluates to `NIL` for some substitution σ then there is some σ' and i such that τ_i/σ' also evaluates to `NIL`. This condition can be made precise in the logic of ACL2 by extending the notion of evaluators discussed in Section 2.4 from terms to clauses. Before describing the extension, we will assume that the ACL2 ground zero theory GZ contains two functions *disjoin* and *conjoin* axiomatized as shown in Figure 1. Informally, the axioms specify how to interpret objects representing clauses and clause lists. For instance, the function *disjoin* specifies that the interpretation of a clause (τ_0 τ_1 τ_2) is the same as the interpretation of (if τ_0 T (if τ_1 T (if τ_2 T NIL)))), which represents the disjunctions of the terms τ_0, τ_1, and τ_2.

Based on these axioms, we can formalize the correctness of clause processors by defining an evaluation semantics for clauses. In particular, assume that *ev* is an evaluator for the single function *if*. Thus *ev*(*list*(`if`, τ_0, τ_1, τ_2), a) stipulates how the term "*if* τ_0 *then* τ_1 *else* τ_2" can be evaluated. For any theory $\mathcal{T}$ (obtained by extending GZ

[2]The formal definition of a clause processor is somewhat more complex. In particular, it can optionally take as argument the current ACL2 state among others, and return, in addition to the list of clauses, an error message and possibly a new ACL2 state. We will ignore such details in this paper.

$$\boxed{\begin{array}{l} \quad \textit{term-listp}(C) \wedge \textit{alistp}(a) \wedge (\textit{ev}(\textit{disjoin}(C), a) = *\texttt{NIL}*) \\ \Rightarrow \\ \quad \textit{ev}(\textit{conjoin}(\textit{tool0}(\texttt{args}, C)), \textit{tool0-env}(\texttt{args}, C, a)) = *\texttt{NIL}* \end{array}}$$

Figure 2: Correctness condition for clause processors. Here *ev* is assumed to be an evaluator for *if*, and `args` represents the remaining arguments of *tool0* (in addition to clause C). The predicates *term-listp* and *alistp* are axiomatized in GZ such that (i) *term-listp*(x) returns a Boolean, which is `T` if and only if x is an object in the ACL2 universe representing a well-formed list of terms (and hence a clause), and (ii) *alistp*(a) returns a Boolean, which is `T` if and only if a is a well-formed association list.

via the extension principles), a clause processor function *tool0*$(\texttt{args}, C)$ will be said to be *legal* in $\mathcal{T}$ if there exists a function *tool0-env* in $\mathcal{T}$ such that the formula shown in Figure 2 is a theorem. The function *tool0-env* returns an association list like σ' in our informal example above: it potentially modifies the original association list to respect any generalization being performed by *tool0*. Note that a weaker theorem would logically suffice, replacing the use of the association list *tool0-env*$(\texttt{args}, c, a)$ by an existentially quantified variable.

A theorem of the form shown in Figure 2 can be tagged with the new rule class for clause processors, instructing ACL2 to use the function *tool0* as a new verified external tool. Theorem 1 below, based on the "Essay on Correctness of Meta Reasoning" comment in the ACL2 sources, guarantees that the above condition is sufficient for the soundness of using *tool0* to transform goal conjectures.

Theorem 1 *Let $\mathcal{T}$ be an ACL2 theory for which tool0 is a legal clause processor, and let tool0 return a list L_C of clauses given an input clause C. If each clause in L_C is provable in $\mathcal{T}$, then $\mathcal{C}$ is also provable in $\mathcal{T}$.*

Proof: The theorem is a simple consequence of the following lemma, given the correctness condition shown in Figure 2. ■

Lemma 1 *Let τ be a term with free variables $v_0, \ldots, v_n$, ev an evaluator for the function symbols in τ, and e a list of cons pairs of the form* `(⟨'v0, '`τ_0`⟩ ... ⟨'vn, '`τ_n`⟩)`, *where* `'`v_i *and* `'`τ_i *are internal representation of v_i and τ_i respectively. Let σ be a substitution mapping each v_i to τ_i, and let* `'`τ *be the internal representation of the term τ. Then the following formula is a theorem: $\textit{ev}('\tau, e) = \tau/\sigma$.*

Proof: An easy induction on the structure of term τ. ■

The simplicity of the above proof might belie some of the subtleties involved. For instance, recall that each ACL2 theory $\mathcal{T}$ is a conservative extension of GZ. Furthermore, note that theorems whose proofs use an invocation of *tool0* often do not involve the function symbols occurring in the definition of the function *tool0* itself. For instance, assume that *tool0* is a simple clause generalizer that replaces each occurrence of a specific subterm in a clause by a free variable not present in the original clause. Such a function can be invoked for generalization in the proof of a formula Φ although Φ might not contain any occurrence of *tool0*. On the completion of a successful proof of Φ, can we then mark *tool0* as local? The answer is in general "no", since Theorem 1 only guarantees provability of the clause input to the clause processor from those returned *in the theory in which the clause processor is legal*. In particular such a theory must contain the definitions of the function symbols being manipulated by *tool0*, and for this it suffices that *tool0* not be marked local. In fact a soundness bug in a previous but very recent release of ACL2 occurred in an analogous context for meta rules, due to ACL2's previous inability to track the fact that the theory in which such rules are applied indeed included the definitions supporting the corresponding evaluators.

4 Basic Unverified External Tools

Verified clause processors are useful when the user intends to augment the reasoning engine of ACL2 with mechanically checked code for customized clause manipulation. However, more often, we want to manipulate goal conjectures using a tool that is external to the theorem prover, for instance a state-of-the-art Boolean satisfiability solver or model checker. In this section, we will consider an extension of the mechanisms to incorporate such tools. In the next section we will present additional constructs to facilitate integration with more general tools.

Our interface for unverified tools involves extending the theorem prover with a new event that enables ACL2 to recognize some function *tool1* defined by the user as an *unverified* clause processor. Here the function *tool1* might be implemented using *program mode* and might also invoke arbitrary executable code using ACL2's system call interface (cf. Section 2.3). The effect of the event in subsequent proof search with ACL2 is the same as if *tool1* were introduced as a verified clause processor: hints can be used to invoke the function for manipulating terms arising during proofs.

Suppose an unverified tool *tool1* simplifies a clause in the course of proving some goal conjecture. What guarantees should an implementor of *tool1* provide (and must the user trust) in order to claim that the goal conjecture is indeed a theorem? In this simple case, a sufficient guarantee is that there is a theory $\mathcal{T}$ containing the definition of *tool1* and appropriate evaluators such that the formula analogous to the one shown in Figure 2 in the previous section for *tool1* is a theorem of $\mathcal{T}$. The soundness of the use of *tool1* then follows from Theorem 1.

Since the invocation of an unverified tool for simplifying ACL2 conjectures carries a logical burden, the event introducing such tools provides two constructs, namely (i) a *tag* for the user of the tool to acknowledge this burden, and (ii) a concept of *supporters* for the tool developer to implement the tool in a way as to be able to guarantee that the logical restrictions are met. We now explain these two constructs.

The tag associated with an event installing an unverified tool *tool1* is a symbol (the default value being the name of the tool itself), which must be used to acknowledge that the soundness of any theorem proven by an application of *tool1* depends on the implementor of *tool1* satisfying the logical guarantees above. The certification of any book that contains an event installing an unverified clause processor (or hereditarily includes such a book, even locally) requires the user to tag the certification command with the name of the tags introduced with the event. Note that technically the mere act of installing an unverified tool does not introduce any unsoundness; the logical burden expressed above pertains to the *use* of the tool. Nevertheless, our decision to insist that the certification of any book with an *installation* of an unverified tool (whether subsequently used or not) to be tagged is governed by implementation convenience. Recall that the local incompatibility check (that is, the second pass of a book certification) skips proofs, and thereby ignores the hints provided during the proof process. By "tracking" the installation rather than the application of an unverified clause processor, we disallow the possibility of a user certifying a book that *locally* introduces an unverified tool and uses it for simplifying some formulas, without acknowledging the application of the tool.

Finally we turn to *supporters*. This construct enables a tool developer to provide the guarantee outlined above in the presence of local events. To understand why this construct is necessary, consider the following scenario. Suppose a developer creates a book (say, `book1`) in which the function f is introduced *locally* with the following definitional axiom:

Local Definitional Axiom.
$f(x) = x$

Suppose further that `book1` also installs an unverified clause processor *tool1*. Assume that the definition of *tool1* does not involve invocation of f, but it replaces terms of the form $f(\tau)$ with τ; thus the correctness of *tool1* depends on the intended definition of f. However, if an ACL2 session is extended by including `book1`, then the extended session contains the definition of *tool0* tagged as an unverified clause processor, but does not contain the (local) definition of f. Thus we can write another book (say, `book2`) that includes `book1` and then provides a new definition of f, for instance the following:

Definitional Axiom.
$f(x) = \mathit{cons}(x, x)$

We now are working in a theory in which *tool1* may be used to perform term manipulations that are completely unjustified by the current definition of f, thus invalidating any guarantee provided by the implementor of *tool1*.

In general, then, suppose that a tool has built-in knowledge about some function symbols. The tool implementor cannot meet the logical burden expressed above unless the user of the tool is required to include the axioms that have been introduced for those function symbols. The *supporters* construct of the event installing unverified clause processors provides a way for the implementor to insist that such axioms are present, by listing the names of axiomatic events (typically function symbols that name their definitions, e.g., f in the example above). We will refer to these events as the *supporting events* for the clause processor. Whenever ACL2 encounters an event installing a function *tool1* as an unverified clause processor with a non-empty list of supporters, it will check that *tool1* and all of the supporting event names are already defined.

5 Templates and Generalized External Tools

The view above of unverified tools is that if a clause processor replaces some clause with a list of clauses then the provability of the resulting clauses implies the provability of the original clause. A clause processor is thus an efficient procedure for assisting in proofs of theorems that could, in principle, have been proven from the axiomatic events of the current theory. This simple view is sufficient in most situations; for instance, one can use it to connect ACL2 with a Boolean satisfiability solver that checks if a propositional formula is a tautology. However, some ACL2 users have found the necessity to use more sophisticated tools that implement their own theory. We will now discuss an extension to the ACL2 logic that facilitates connection with such tools.

To motivate the need for such tools, assume that we wish to prove a theorem about some hardware design. Most such designs are written in a Hardware Description Language (HDL) such as VHDL or Verilog. One way of formalizing such designs is to define a semantics of the HDL in ACL2, possibly by defining a formal interpreter for the language. However, defining such an interpreter is typically extremely complex and labor-intensive. On the other hand, there are several model checkers available which can parse designs written in VHDL or Verilog. An alternative is merely to *constrain* some properties of the interpreter and use a combination of theorem proving and model checking in the following manner:

- Establish low-level properties of parts of a design using model checkers or other decision procedures.

- Use the theorem prover to compose the properties proven by the model checker together with the constrained properties of the interpreter to establish the correctness of the design.

The above approach has shown promise in scaling formal verification to industrial designs. For instance, Sawada and Reeber [SR06] have recently verified an industrial VHDL floating-point multiplier using a combination of ACL2 and an IBM internal verification tool called SixthSense [MBP$^+$04]. They introduce two functions, ***sigbit*** and ***sigvec***, with the following assumed semantics:

- ***sigbit***(e, s, n, p) returns a bit corresponding to the value of bit signal s of a VHDL design e at cycle n and phase p.

- ***sigvec***(e, s, l, h, n, p) returns a bit vector corresponding to the bit-range between l and h of s for design e at cycle n and phase p.

In ACL2 these two functions are constrained only to return a bit and bit-vector respectively. The key properties of the different multiplier stages are proven using SixthSense. For instance, one of the properties proven is that ***sigvec*** when applied to (i) a constant `C` representing the multiplier design, (ii) a specific signal `s` of the design, (iii) two specific values `lb` and `hb` corresponding to the bit-width of s, and (iv) a specific cycle and phase, returns the sum of two other bit vectors at the previous cycle; this corresponds to one stage of the Wallace-tree decomposition implemented by the multiplier. All such theorems are then composed by ACL2 to verify that the multiplier, when provided two vectors of the right size, produces their product after 5 cycles.

How do we support this verification approach? Note that the property above is *not* provable from the constraints on the associated functions alone (namely ***sigvec*** returns a bit vector). Thus if we use encapsulation to constrain ***sigvec*** and posit the property as a theorem then functional instantiation can derive an inconsistency. The problem is that the property is provable from the constraints *together with* axioms about *sigvec* that are unknown to ACL2 but assumed to be accessible to SixthSense.

Our solution to the above is to extend the extension principles of ACL2 with a new principle called *encapsulation templates* (or simply *templates*). Function symbols introduced via templates are constrained functions just like those introduced via the encapsulation principle, and the soundness of extending an ACL2 theory is analogously guaranteed by exhibiting a local witness satisfying the constraints. However, there is one significant distinction between encapsulation principle and templates: the constraints introduced are marked *incomplete*, acknowledging that they might not encompass all the constraints on the functions. ACL2 therefore disallows functional instantiation of theorems by substituting for functions introduced via templates.

The use of template events facilitates integration of ACL2 with tools like SixthSense above. Suppose that we wish to connect ACL2 with an unverified tool *tool1* that implements a theory that we do not wish to define explicitly in ACL2. We then use a template event to introduce the function symbols (say f and g) regarding which the theory of the clause processor contains additional axioms. Finally we introduce *tool1* as an unverified clause processor, marking f and g as supporting events.

We now explain the logical burden for the developer of such a connection. Assume that an ACL2 theory $\mathcal{T}$ is extended by a template event E, and suppose that the supporting events for *tool1* mention some function introduced by E. Then the developer of *tool1* must guarantee that it is possible, in principle, to introduce f and g via the encapsulation principle (which we will refer to as the "promised" encapsulation E_P of the functions) such that the following conditions hold:

1. The constraints in E_P include the constraints in E.
2. E_P does not introduce any additional function symbols other than those introduced by E.
3. E_P is admissible in theory $\mathcal{T}$.
4. For any extension $\mathcal{T}_0$ of $\mathcal{T}$ together with the constraints in E_P, if one can invoke *tool1* to reduce some clause C to a list of clauses L_C then if each clause of L_C is first-order provable (with induction) in $\mathcal{T}_0$ then C must be provable in $\mathcal{T}_0$.

Furthermore, in order to make logical guarantees regarding ACL2 sessions that contain events corresponding to *several* unverified external tools, ACL2 enforces the following "disjointness condition": a template event may not be extended to a promised encapsulate by two different clause processors. Thus, when an unverified clause processor installation event has a supporting event name, f, such that f is a function symbol that had been introduced by a template, it is required that no unverified clause processor has been previously installed in the current ACL2 session that has a supporting event name that is a function symbol introduced in the same template. This makes it possible to view the event as essentially the (unique) promised encapsulation whose existence is guaranteed by the implementor of the tool. Note that condition 2 above is necessary for this purpose to preclude the possibility that the theory implemented by different external tools might have conflicting implicit axioms in their promised encapsulations for function symbols not introduced by the template.

With these conditions, we can make the following informal claim for an ACL2 session which includes templates together with the use of unverified clause processors:

> Perform the following transformation in sequence to each template event E in the session. If there is a tool *tool1* whose supporting events mention a function symbol introduced by E then replace E with the encapsulation E_P promised by the developer of *tool1*. Otherwise extend E to an arbitrary admissible encapsulate. (Note that at least one such extension exists, namely one in which no additional constraint is introduced.) Then every alleged theorem in the session is in fact derivable in first-order logic (with induction) from the axiomatic events in the session produced after this transformation.

The informal claim above can be made precise by formalizing the notion of an ACL2 session. Kaufmann and Moore [KM01] describe such a formalization where a valid session is modeled as a *chronology*, inductively defined as a sequence of events that is either (i) the empty sequence, or (ii) constructed from a sequence by introducing one of the legal ACL2 events such as commands for introducing new functions, and proving theorems. We omit that description here and refer the reader to their paper [KM01] for details. For this paper, we point out that given a careful inductive characterization of a session as a chronology, it is easy to see that an ACL2 session transformed as above really corresponds to a chronology. The basic observation is that for any chronology in which no function introduced by an encapsulation is functionally instantiated, the encapsulation may be strengthened and the result is still a chronology. The proof is by induction on the formation of chronologies, and each proof obligation encountered in the inductive step is discharged against the possibly stronger theory.

6 Related Work

The importance of allowing the hooking up of external tools has been widely recognized in the theorem proving community. Some early ideas for connecting different theorem provers are discussed in a proposal for so-called "interface logics" [Gut91], with the goal to connect automated reasoning tools by defining a single logic L such that the logics of the individual tools can be viewed as sub-logics of L. More recently, with the success of model checkers and Boolean satisfiability solvers, there has been significant work connecting such tools with interactive theorem proving. The PVS theorem prover provides connections with several decision procedures such as model checkers and SAT solvers [RSS95, Sha01]. The Isabelle theorem prover [Pau] uses unverified external tools as *oracles* for checking formulas as theorems during a proof search; this mechanism has been used to integrate model checkers and arithmetic decision procedures with Isabelle [MN95, BF00]. Oracles are also used in the HOL family of higher order logic theorem provers [GM93]; for instance, the PROSPER project [DCN$^+$00] uses the HOL98 theorem prover as a uniform and logically-based coordination mechanism between several verification tools. The most recent incarnation of this family of theorem provers, HOL4, uses an external oracle interface to decide large Boolean formulas through connections to state-of-the-art BDD and SAT-solving libraries [Gor02], and also uses that oracle interface to connect HOL4 with ACL2 as discussed in the next section.

The primary basis for interfacing external tools with theorem provers for higher-order logic (specifically HOL and Isabelle) involves the concept of "theorem tagging", introduced by Gunter for HOL90 [Gun98]. The idea is to introduce a tag *in the logic* for each oracle and view a theorem certified by the oracle as an implication with the tag corresponding to the certifying oracle as a hypothesis. This approach enables tracking of dependencies on unverified tools at the level of individual theorems. In contrast, our approach is designed to track such dependencies at the level of files, that is, ACL2 books. Our coarser level of tracking is at first glance unfortunate: if a book contains some events that depend on such tools and others that do not, then the entire book is "tainted" in the sense that its certification requires an appropriate acknowledgement for the tools. We believe that this will not prove to be an issue in practice, as ACL2 users typically find it easy to move events between books. On the positive side, it is simpler to track a single event introducing an external tool rather than uses of such an event, especially since hints are ignored when including previously certified books. As an aside, we note that a very general tagging mechanism is under development for ACL2, serving as a foundation in particular for tagging of unverified clause processors.

There has also been work on using an external tool to search for a proof that can then be checked by the theorem prover without assistance from the tool. Hurd [Hur02] describes such an interface connecting HOL with first-order logic. McCune and Shumsky [MS00] present a system called Ivy which uses Otter to search for first-order proofs of equational theories and then invokes ACL2 to check such proof objects. Meng and Paulson [MP04] interface Isabelle with a resolution theorem prover.

Several ACL2 users have integrated external tools with ACL2; but without the disciplined mechanisms of this paper, such integration has essentially involved implementation hacks on the ACL2 source code. Ray, Matthews, and Tuttle integrate ACL2 with SMV [RMT03]. Reeber and Hunt connect ACL2 with the Zchaff satisfiability solver [RH06], and Sawada and Reeber provide a connection with SixthSense [SR06]. Manolios and Srinivasan connect ACL2 with UCLID [MS04, MS05].

7 Conclusion and Future Work

Different deduction tools bring in different capabilities to formal verification. A strength of general purpose theorem provers compared to many tools based on decision procedures is in the expressive power of the logic, which enables succinct definitions. Automatic decision procedures provide more automated proof procedures for decidable theories. Several ACL2 users have requested ways to connect ACL2 with automated decision procedures. We believe that the mechanisms described in this paper will provide a disciplined way of using ACL2 with other tools with a clear specification of the expectations from the tool in order to guarantee soundness of the ACL2 session. Furthermore, we believe that verified clause processors will provide a way for the user to control a proof more effectively without relying on ACL2's heuristics.

We have presented an approach to connecting ACL2 with external deduction tools, but we have merely scratched the surface. It is well-known that developing an effective interface between two or more deduction tools is a complicated exercise [KM92]. It remains to be seen how to effectively decompose theorem proving problems so as to make effective use of clause processors to provide the requisite automation.

Some researchers have criticized our interface on the grounds that developing a connection with an external tool requires significant knowledge of the ACL2 logic. While we acknowledge that our interface requires understanding of that logic, including the term representation, we believe that such requirement is necessary for any developer interested in developing connections between formal tools. A connection between different formal tools must involve a connection between two logics, and the builder of such connection must understand both the logics, including the legal syntax of terms, and the axioms and rules of inferences. It should be noted that the logic of ACL2 is perhaps more complex than many others, principally because it offers proof structuring mechanisms by enabling the user to mark events as *local*. This complexity manifests itself in the interface; constructs such as *supporters* are provided essentially to enable the tool implementor to provide logical guarantees in the presence of local events. However, we believe that with these constructs it will be possible for the tool developers to implement connections with ACL2 with reasonable understanding of the theorem prover.

Finally, note that the restrictions for the tool developers that we have outlined preclude certain external deduction tools. For instance, there has been recent work connecting HOL with ACL2 [GHKR06a, GHKR06b]; the approach there has been for a HOL user to make use of ACL2's proof automation and fast execution capabilities. It might be of interest to the ACL2 user to take advantage of HOL's expressive power as well. We are working on extending the logical foundations of ACL2 to facilitate such a connection. The key idea is that the ACL2 theorem prover might be viewed as a theorem prover for the HOL logic. If the view is accurate then it will be possible for the user of ACL2 to prove some formulas in HOL and use them in an ACL2 session, claiming that the session essentially reflects a HOL session mirrored in ACL2.

Acknowledgements

The authors thank Peter Dillinger, Robert Krug, Pete Manolios, John Matthews, and Rob Sumners for comments and feedback.

References

[BF00] D. Basin and S. Friedrich. Combining WS1S and HOL. In D. M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000.

[BH97] B. Brock and W. A. Hunt, Jr. Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In *Proceedings of the* 1997 *International Conference on Computer Design: VLSI in Computers & Processors (ICCD 1997)*, pages 31–36, Austin, TX, 1997. IEEE Computer Society Press.

[BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them Correct and Using Them Efficiently as New Proof Procedure. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, London, UK, 1981.

[DCN$^+$00] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. F. Melham. The PROSPER toolkit. In S. Graf and M. Schwartbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for Constructing Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 78–92, Berlin, Germany, 2000. Springer-Verlag.

[FKR$^+$02] A. Flatau, M. Kaufmann, D. F. Reed, D. Russinoff, E. W. Smith, and R. Sumners. Formal Verification of Microprocessors at AMD. In M. Sheeran and T. F. Melham, editors, *4th International Workshop on Designing Correct Circuits (DCC 2002)*, Grenoble, France, April 2002.

[GHKR06a] M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, and J. Reynolds. An embedding of the ACL2 logic in HOL. In P. Manolios and M. Wilding, editors, *6th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, Seattle, WA, August 2006.

[GHKR06b] M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, and J. Reynolds. An integration of HOL and ACL2. In A. Gupta and P. Manolios, editors, *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006)*, San Jose, CA, November 2006. Springer-Verlag.

[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.

[Gor02] M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, 2002.

[GRW04] D. A. Greve, R. Richards, and M. Wilding. A Summary of Intrinsic Partitioning Verification. In M. Kaufmann and J S. Moore, editors, *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, November 2004.

[Gun98] E. L. Gunter. Adding External Decision Procedures to HOL90 Securely. *Lecture Notes in Computer Science*, 1479:143–152, 1998.

[Gut91] J. D. Guttman. A Proposed Interface Logic for Verification Environments. Technical Report M-91-19, The Mitre Corporation, March 1991.

[Hur02] J. Hurd. An LCF-Style Interface between HOL and First-Order Logic. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE 2002)*, volume 2392 of *LNCS*, pages 134–138. Springer-Verlag, 2002.

[KM92] M. Kaufmann and J S. Moore. Should We Begin a Stanrdization Process for Interface Logics? Technical Report 72, Computational Logic Inc. (CLI), January 1992.

[KM94] M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic Incorporated (CLI), 1717 West Sixth Street, Suite 290, Austin, TX 78703, 1994.

[KM97] M. Kaufmann and J S. Moore. A Precise Description of the ACL2 Logic. See URL `http://www.cs.utexas.edu/users/moore/-publications/km97.ps.gz`, 1997.

[KM01] M. Kaufmann and J S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[KM06] M Kaufmann and J S. Moore. ACL2 home page, 2006. See URL `http://-www.cs.utexas.edu/users/moore/acl2`.

[KMM00] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.

[MBP+04] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable Automated Verification via Exper-System Guided Transformations. In A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 217–233. Springer-Verlag, 2004.

[MLK98] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Kernel of the AMD5K86 Floating-point Division Algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.

[MN95] O. Müller and T. Nipkow. Combining Model Checking and Deduction of I/O-Automata. In E. Brinksma, editor, *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1995)*, volume 1019 of *LNCS*, Aarhus, Denmark, May 1995. Springer-Verlag.

[MP04] J. Meng and L. C. Paulson. Experiments on Supporting Interactive Proof Using Resolution. In D. A. Basin and M. Rusinowitch, editors, *Proceedings of the 2nd International Joint Conference on Computer-Aided Reasoning (IJCAR 2004)*, volume 3097 of *LNCS*, pages 372–384, 2004.

[MS00] W. McCune and O. Shumsky. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In P. Manlolios, M. Kaufmann, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 217–230. Kluwer Academic Publishers, Boston, MA, June 2000.

[MS04] P. Manolios and S. Srinivasan. Automatic Verification of Safety and Liveness of XScale-Like Processor Models Using WEB Refinements. In *Design, Automation and Test in Europe (DATE 2004)*, pages 168–175, Paris, France, 2004. IEEE Computer Society Press.

[MS05] P. Manolios and S. Srinivasan. Refinement Maps for Efficient Verification of Processor Models. In *Design, Automation and Test in Europe (DATE 2005)*, pages 1304–1309, Munich, Germany, 2005. IEEE Computer Society Press.

[Pau] L. Paulson. The Isabelle Reference Manual. See URL `http://www.cl.-cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle2003/doc/ref.pdf`.

[RF00] D. Russinoff and A. Flatau. RTL Verification: A Floating Point Multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA, June 2000. Kluwer Academic Publishers.

[RH06] E. Reeber and W. A. Hunt, Jr. A SAT-Based Decision Procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA). In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Computer-Aided Reasoning (IJCAR 2006)*, volume 4130 of *LNAI*, pages 453–467, 2006.

[RMT03] S. Ray, J. Matthews, and M. Tuttle. Certifying Compositional Model Checking Algorithms in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and J S. Moore, editors, *4th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, Boulder, CO, July 2003.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An Integration of Model-Checking with Automated Proof Checking. In P. Wolper, editor, *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.

[Rus98] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-point Multiplication, Division, and Square Root Instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, December 1998.

[SH97] J. Sawada and W. A. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[Sha01] N. Shankar. Using Decision Procedures with Higher Order Logics. In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the 14th International Conference on Theorem Proving in Higher-Order Logics (TPHOLS 2001)*, volume 2152 of *LNCS*, pages 5–26. Springer-Verlag, 2001.

[SR06] J. Sawada and E. Reeber. ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool. In A. Gupta and P. Manolios, editors, *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2006)*, San Jose, CA, November 2006. Springer-Verlag.

Efficiently Checking Propositional Resolution Proofs in Isabelle/HOL

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

Abstract

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

1 Introduction

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

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

Formal verification is an important application area of interactive theorem proving. Problems in verification can often be reduced to Boolean satisfiability (SAT), and recent SAT solver advances have made this approach feasible in practice. Hence the performance of an interactive prover on propositional problems may be of significant practical importance. In this paper we describe the integration of zChaff [MMZ+01] and MiniSat [ES04], two leading SAT solvers, with the Isabelle/HOL [NPW02] prover.

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

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

2 System Description

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

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

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

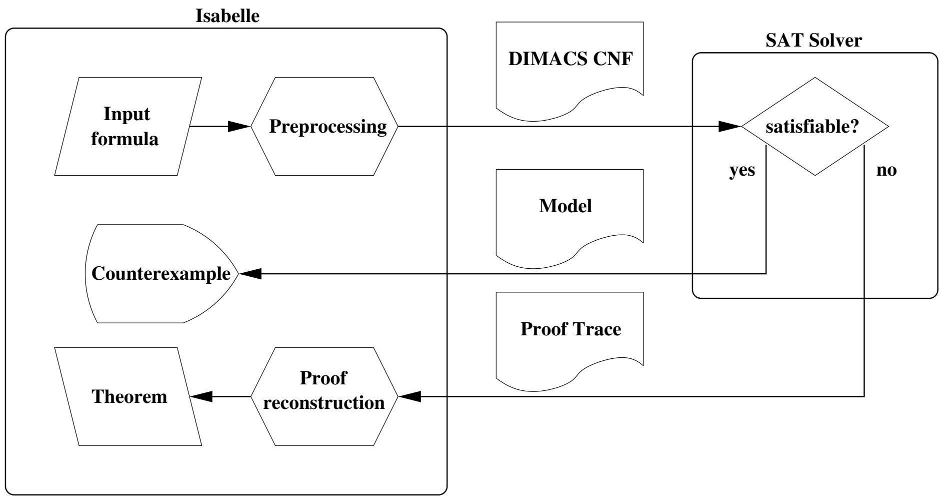


Figure 1: Isabelle – SAT System Architecture

found by zChaff and MiniSat.

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

2.1 Preprocessing

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

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

Note that it is not sufficient to convert ϕ into an equivalent formula ϕ^* in CNF. Rather, we have to *prove* this equivalence inside Isabelle/HOL. The result is not a single formula, but a theorem of the form $\vdash \phi = \phi^*$. The fact that our CNF transformation must be proof-producing leaves some potential for optimization. One could implement a non proof-producing (and therefore much faster) version of the same CNF transformation, and use it for preprocessing instead. Application of the proof-producing version would then be necessary only if the SAT solver has in fact shown a formula to be unsatisfiable. The total runtime on provable formulae would increase slightly, as the CNF transformation needed to be done twice – first without, later with proofs. Preprocessing times for unprovable formulae however should improve.

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

2.2 Proof Reconstruction

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

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

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

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

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

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

2.2.1 zChaff's Proof Trace Format

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

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

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

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

2.2.2 MiniSat's Proof Trace Format

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

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

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

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

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

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

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

2.2.3 Proof Reconstruction

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

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

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

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

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

Let us use $[\![A_1; \ldots; A_n]\!] \Longrightarrow B$ as a short hand for $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$ (with implication associating to the right). In [FMM+06], each clause $p_1 \vee \ldots \vee p_n$ is encoded as an implication $\overline{p_1} \Longrightarrow \ldots \Longrightarrow \overline{p_n} \Longrightarrow \text{False}$ (where $\overline{p_i}$ denotes the negation normal form of $\neg p_i$, for $1 \leq i \leq n$), and turned into a separate theorem

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

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

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

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

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

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

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

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

and

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

We then instantiate a previously proven theorem

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

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

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

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

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

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

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

Another optimization, related not to the representation of individual clauses, but to the overall proof structure, is perhaps more obvious and has been present in our implementations since the beginning. zChaff and MiniSat, during proof search, may generate many clauses that are ultimately not needed to derive the empty clause. Instead of replaying the whole proof trace in chronological order, we perform "backwards" proof reconstruction, starting with the identifier of the empty clause, and recursively proving the required resolvents using depth-first search.

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

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

2.3 A Simple Example

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

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

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

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

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

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

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

$$\dfrac{\dfrac{\neg x_2 \lor x_3 \qquad \dfrac{x_1 \lor x_2 \qquad \neg x_1 \lor x_2}{x_2}}{x_3} \qquad \dfrac{\neg x_2 \lor \neg x_3 \qquad \dfrac{x_1 \lor x_2 \qquad \neg x_1 \lor x_2}{x_2}}{\neg x_3}}{\bot}$$

Figure 2: Resolution Proof found by zChaff

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

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

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

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

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

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

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

$$\dfrac{\dfrac{\neg x_1 \vee x_2 \qquad \dfrac{\neg x_2 \vee x_3 \qquad \neg x_2 \vee \neg x_3}{\neg x_2}}{\neg x_1} \qquad \dfrac{x_1 \vee x_2 \qquad \dfrac{\neg x_2 \vee x_3 \qquad \neg x_2 \vee \neg x_3}{\neg x_2}}{x_1}}{\bot}$$

Figure 3: Resolution Proof found by MiniSat

Problem Representation	Proof Reconstruction (zChaff)
Naive HOL	726.5
Separate Clauses	7.8
Sequent	1.2
CNF Sequent	0.5

Table 1: Runtimes (in seconds) for MSC007-1.008

3 Evaluation

In [Web05a], we compared the performance of our approach, using the naive HOL problem representation, to that of Isabelle's existing automatic proof procedures on all 42 problems contained in version 2.6.0 of the TPTP library [SS98] that have a representation in propositional logic. The problems were negated, so that unsatisfiable problems became provable. The benchmarks were run on a machine with a 3 GHz Intel Xeon CPU and 1 GB of main memory.

19 of these 42 problems are rather easy, and were solved in less than a second each by both the existing procedures and the SAT solver approach. On the remaining 23 problems, zChaff proved to be clearly superior to Isabelle's built-in proof procedures. zChaff solved all problems in less than a second, and proof reconstruction in Isabelle/HOL took a few seconds at most for all but one problem: with the naive HOL representation, the proof for problem MSC007-1.008 was reconstructed in just over 12 minutes.

To give an impression of the effect that the different clause representations discussed in Section 2.2.3 have on performance, Table 1 shows the different times required to prove problem MSC007-1.008. The proof found by zChaff for this problem has 8,705 resolution steps. MiniSat finds a proof with 40,790 resolution steps for the same problem, which is reconstructed in about 3.8 seconds total with the sequent representation, and in 1.1 seconds total with the CNF sequent representation. The times to prove the other problems from the TPTP library have decreased in a similar fashion and are well below one second each now.

This enables us to evaluate the performance on some significantly larger problems, taken from the SATLIB library [HS00]. These problems do not only push Isabelle's inference kernel to its limits, but also other parts of the prover. While the smaller TPTP problems were converted to Isabelle's input syntax by a Perl script, this approach turns out to be infeasible for the larger SATLIB problems. The Perl script still works fine, but Isabelle's parser (which was mainly intended for small, hand-crafted terms) is unable to parse the resulting theory 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 little parser, which builds ML terms directly from

Problem	Variables	Clauses	zChaff (s)	Proof (s)	Resolutions	Total (s)
c7552mul.miter	11282	69529	73	70	252200	145
6pipe	15800	394739	167	321	268808	512
6pipe_6_ooo	17064	545612	308	2575	870345	3179
7pipe	23910	751118	495	1132	357136	1768

Table 2: Runtimes (in seconds) for SATLIB problems, zChaff

Problem	Variables	Clauses	MiniSat (s)	Proof (s)	Resolutions	Total (s)
c7552mul.miter	11282	69529	25	49	908231	106
6pipe	15800	394739	**x**	—	—	—
6pipe_6_ooo	17064	545612	**x**	—	—	—
7pipe	23910	751118	**x**	—	—	—

Table 3: Runtimes (in seconds) for SATLIB problems, MiniSat

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

Statistics for four unsatisfiable SATLIB problems (chosen among those that were used to evaluate zChaff's performance in [ZM03]) are shown in Tables 2 and 3, for zChaff and MiniSat respectively. The first column shows the time in seconds that it takes the SAT solver to find a proof of unsatisfiability. The second column, "Proof", shows the time in seconds required to replay the proof's resolutions steps in Isabelle/HOL, using the CNF sequent representation of clauses. The third column shows the number of resolution steps performed during proof replay. The last column, "Total", finally shows the total time to prove the problem unsatisfiable in Isabelle, including SAT solving time, proof replay, parsing of input and output files, and any other intermediate pre- and postprocessing. These timings were obtained on an AMD Athlon 64 X2 Dual Core Processor 3800+ with 4 GB of main memory. An **x** indicates that the solver ran out of memory, or that the proof trace file exceeded a size of 2 GB. Needless to say that none of these problems can be solved automatically by Isabelle's built-in proof procedures.

It seems that proof checking in Isabelle/HOL, despite all optimizations that we have implemented, is sometimes about an order of magnitude slower than proof verification with an external checker written in C++ [ZM03]. From Table 2 we conclude that proving unsatisfiability in Isabelle/HOL is by a factor of roughly 2 to 10 slower than using zChaff alone. This additional overhead was to be expected: it is the price that we have to pay for using Isabelle's LCF-style kernel, which is not geared towards propositional logic. However, we also see that proof reconstruction in Isabelle scales quite well with our latest implementation, and that it remains feasible even for large SAT problems.

Comparing the runtimes for problem c7552mul.miter on the proofs found by zChaff and MiniSat, we see that the time taken to reconstruct a proof does not solely depend on the number of resolutions steps. In particular our algorithm for resolving two clauses, as described in Section 2.2.3, is linear in the length (i.e. number of literals) of those clauses. The average length of a clause is about 31.0 for the MiniSat proof, and about 98.6 for the proof found by zChaff. This explains why the zChaff proof, despite its smaller number of resolution steps, takes longer to reconstruct.

4 Related Work

Michael Gordon has implemented *HolSatLib* [Gor01], a library which is now part of the HOL 4 theorem prover. This library provides functions to convert HOL 4 terms into CNF, and to analyze them using a SAT solver. In the case of unsatisfiability however, the user only has the option to trust the external solver. No proof reconstruction takes 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 HOL 4.

Perhaps closer related to our work is the integration of automated first-order provers, in the context of Isabelle recently further 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]. Also Andreas Meier's TRAMP system [Mei00] transforms the output of various automated first-order provers into natural deduction proofs. The main focus of their work 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].

A custom-built SAT solver has been integrated with the CVC Lite system [BB04] by Clark Barrett et al. [BBD03]. While this solver produces proofs that can be checked independently, our work shows that it is possible to integrate existing, highly efficient solvers with an LCF-style prover: the information provided by recent versions of zChaff and MiniSat is sufficient to produce a proof object in a theorem prover, no custom-built solver is necessary.

An earlier version of this work was presented in [Web05a], and improved by Alwen Tiu et al. [FMM$^+$06]. Furthermore Hasan Amjad [Amj06b] has recently integrated a proof-generating version of the MiniSat solver with HOL 4 in a similar fashion. 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 proof procedures that were previously available in Isabelle/HOL. With the help of zChaff or MiniSat, many formulae that were previously out of the scope of Isabelle's built-in tactics can now be proved – or refuted – automatically, often within seconds. Isabelle's 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 several hundred thousand resolution steps. The additional confidence gained by using an LCF-style prover to check the proof obviously comes at a price (in terms of runtime), but it's not nearly as expensive as one might have expected after earlier implementations.

While improving the performance of our implementation, we confirmed an almost self-evident truth: use profiling to see which functions take a lot of time, and focus on improving them – this is were the greatest benefits lie. This was an iterative process. A better implementation would allow us to tackle larger SAT problems, which in turn would uncover new performance bottlenecks. More importantly, we discovered some inefficiencies in the implementation of the Isabelle kernel. (Instantiating a theorem with a term, for example, was linear in the size of the term, rather than in constant time.) 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 negative impact on performance. Subsequently the kernel implementation was modified, and these inefficiencies were removed.

Tuning an implementation to the extend presented here requires a great deal of familiarity with the underlying theorem prover. Nevertheless our results are applicable beyond Isabelle/HOL. Other interactive provers for higher-order logic, e.g. HOL 4 and HOL-Light, use very similar data structures to represent their theorems. Hasan Amjad has confirmed that the CNF sequent representation works equally well in these provers [Amj06b].

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 [Amj06a].

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. This has already been used to integrate haRVey, a Satisfiability Modulo Theories (SMT) prover, 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.

Acknowledgments. The author would like to thank Markus Wenzel for several good ideas and his extensive help with tuning the implementation, and Hasan Amjad for more good ideas and the challenge that he posed. The clause representations used in this paper were suggested to me by various people, including Hasan Amjad, John Harrison, John Matthews, and Markus Wenzel.

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.

[Amj06a] 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.

[Amj06b] Hasan Amjad. A HOL/MiniSat interface. Personal communication, September 2006.

[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.

[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 `http://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 documentation, version 1.0b, June 2001. Available online at `http://www.cl.cam.ac.uk/~mjcg/HolSatLib/HolSatLib.html`.

[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.

[Mat06] John Matthews. ASCII proof traces for MiniSat. Personal communication, August 2006.

[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, Lawrence 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.

[SS98] Geoff Sutcliffe and Christian Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. Available online at `http://www.cs.miami.edu/~tptp/`.

[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.

[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.

Implementing an Instantiation-based Theorem Prover for First-order Logic

Konstantin Korovin
The University of Manchester, England
korovink@cs.man.ac.uk

The basic idea behind instantiation-based theorem proving is to combine clever generation of instances of clauses with satisfiability checking of ground formulas. There are a number of approaches developed and implemented in recent years: Ordered Semantic Hyper Linking of Plaisted and Zhu, Disconnection Calculus of Letz and Stenz implemented in DCTP, Model Evolution Calculus of Baumgartner and Tinelli implemented in Darwin, and instantiation approach of Claessen implemented in Equinox.

One of the distinctive features of the approach we have been developing is a modular combination of ground reasoning with instantiation. In particular, this approach allows us to employ any off-the-shelf propositional satisfiability solver in a general context of first-order reasoning. In our previous work (together with Harald Ganzinger) we developed a theoretical background for this instantiation method and have shown completeness results together with general criteria for redundancy elimination. In order to evaluate the practical applicability, we implemented our method in iProver.

This talk focuses on implementation issues of an instantiation-based theorem prover, based upon our experience with iProver. We show how our abstract framework can be turned into a concrete implementation. We discuss how well-studied technology of implementing resolution theorem provers can be adapted for implementing iProver and issues specific to instantiation. We show some concrete redundancy criteria for instantiation and how they are implemented in iProver. Finally, we discuss some future directions.

A Tableau Decision Procedure for Propositional Intuitionistic Logic

Alessandro Avellone, Guido Fiorino, Ugo Moscato
Dipartimento di Metodi Quantitativi per l'Economia,
Università Milano-Bicocca,Piazza dell'Ateneo Nuovo, 1, 20126 Milano, Italy
{alessandro.avellone,guido.fiorino,ugo.moscato}@unimib.it

November 3, 2006

Abstract

The contribution of this paper consists in some techniques to bound the proof search space in propositional intuitionistic logic. These techniques are justified by Kripke semantics and they are the backbone of a tableau based theorem prover (PITP) implemented in C++ language. PITP and some known theorem provers are compared by the formulas of ILTP v1.1.1 benchmark library. It turns out that PITP is, at the moment, the propositional prover that solves most formulas of the library.

1 Introduction

The development of effective theorem provers for intuitionistic and constructive logics is of interest both for the investigations and applications of such logics to formal software/hardware verification and program synthesis (see i.e. [ML84, Con86, ES99, Men00, FFO02, BC04, AFF$^+$06]).

In this paper we present a strategy and an implementation based on a tableau calculus for propositional intuitionistic logic. Our decision procedure implements the tableau calculus of [AFM04] (this calculus is an enhancement of the calculus given in [Fit69] and it is related to the tableau and sequent calculi of [MMO97, Hud93, Dyc92]).

We introduce some new techniques utilized by our decision procedure to narrow the search space and the width of the proofs. The PSPACE-completeness of intuitionistic validity ([Sta79]) suggests that backtracking and branching cannot be eliminated. In order to improve the time efficiency of the implementations and make them usable, strategies have to be developed to bound backtracking and branching as much as possible.

The optimizations we present are explained by the Kripke semantics for intuitionistic logic. Such semantical techniques are related to the fact that tableau calculi are strictly joined to the semantics of the logic at hand. Loosely speaking, a tableau proof for a formula is the attempt to build a model satisfying the formula. The construction of such a model proceeds by increasing, step by step, the information necessary to define such a model (thus, step by step the accuracy of the model increases). If the proof ends in a contradiction, then there is no model for the formula. Otherwise, a model satisfying the formula is immediately derived from the proof. With this machinery at hand, first we provide a sufficient condition allowing us to stop a tableau proof without loosing the completeness. Then we describe a technique to bound branching on the formulas which only contain conjunctions and disjunctions. Finally we present a technique to deduce the satisfiability of a set of formulas S, when the satisfiability of a set S' and a permutation τ such that $S = \tau(S')$ are known. Such a technique allows us to bound backtracking. Our technique to bound backtracking is different from the semantical technique provided in [Wei98].

Besides the strategy and its completeness, in the final part of the paper we present some experimental results on the implementation PITP. PITP is written in C++ and it is tested on the propositional part of ILTP v1.1.1 benchmark library ([ROK06]). Of 274 propositional benchmarks contained in ILTP v1.1.1, PITP decides 215 formulas including 13 previously unsolved problems within the time limit of ten minutes. To give the reader more elements to evaluate PITP strategies, comparisons with different versions of PITP are provided.

2 Notation and Preliminaries

We consider the propositional language $\mathcal{L}$ based on a denumerable set of propositional variables or atoms $\mathcal{PV}$ and the logical connectives $\neg, \wedge, \vee, \rightarrow$. (Propositional) Kripke models are the main tool to semantically characterize (propositional) intuitionistic logic **Int** (see [CZ97] and [Fit69] for the details). A Kripke model for $\mathcal{L}$ is a structure $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$, where $\langle P, \leq, \rho \rangle$ is a poset with minimum ρ and $\Vdash$ is the forcing relation, a binary relation between elements α of P and p of $\mathcal{PV}$ such that $\alpha \Vdash p$ and $\alpha \leq \beta$ imply that $\beta \Vdash p$. The forcing relation is extended in a standard way to arbitrary formulas of $\mathcal{L}$ as follows:

1. $\alpha \Vdash A \wedge B$ iff $\alpha \Vdash A$ and $\alpha \Vdash B$;
2. $\alpha \Vdash A \vee B$ iff $\alpha \Vdash A$ or $\alpha \Vdash B$;
3. $\alpha \Vdash A \rightarrow B$ iff, for every $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash A$ implies $\beta \Vdash B$;
4. $\alpha \Vdash \neg A$ iff for every $\beta \in P$ such that $\alpha \leq \beta$, $\beta \Vdash A$ does not hold.

We write $\alpha \nVdash A$ when $\alpha \Vdash A$ does not hold.

It is easy to prove that for every formula A, if $\alpha \Vdash A$ and $\alpha \leq \beta$, then $\beta \Vdash A$. A formula A *is valid in a Kripke model* $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ if and only if $\rho \Vdash A$. It is well known that **Int** coincides with the set of formulas valid in all Kripke models.

If we consider Kripke models $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ such that $|P| = 1$ we get classical models for (propositional) classical logic **Cl**. Classical models are usually seen as functions σ from $\mathcal{PV}$ to $\{true, false\}$. Given a formula A and a model σ, we use $\sigma \models A$ with the usual meaning of satisfiability. Finally, given a set S, the set $\mathcal{PV}(S)$ denotes the elements of $\mathcal{PV}$ occurring in S.

In the following presentation, we give a brief overview of the tableau calculus **Tab** of [AFM04] which is implemented by our decision procedure. The rules of the calculus are given in Table 1. The calculus works on signed well formed formulas (swff for short), where a swff is a (propositional) formula prefixed with a sign $\mathbf{T}$, $\mathbf{F}$ or $\mathbf{F_c}$. Given a Kripke model $\underline{K} = \langle P,\leq,\rho,\Vdash\rangle$, a world $\alpha \in P$, a formula A and a set of swffs S, the meaning of the signs is as follows:

- $\alpha \rhd \mathbf{T}A$ (α realizes $\mathbf{T}A$) iff $\alpha \Vdash A$;
- $\alpha \rhd \mathbf{F}A$ iff $\alpha \nVdash A$;
- $\alpha \rhd \mathbf{F_c}A$ iff $\alpha \Vdash \neg A$;
- $\alpha \rhd S$ iff α realizes every swff in S.
- $\underline{K} \rhd S$ iff $\rho \rhd S$.

A proof table (or proof tree) for S is a tree, rooted with S and obtained by the subsequent application of the rules of the calculus. As an example, let $\Gamma = \{\mathbf{T}(A \wedge B), \mathbf{T}(B \wedge C), \mathbf{F}(A \vee B)\}$. With " rule $\mathbf{T}\wedge$ applies to Γ taking $H \equiv \mathbf{T}(A \wedge B)$ as main swff" we mean that $\mathbf{T}\wedge$ applies to Γ as $\frac{\Gamma}{\Gamma\setminus\{\mathbf{T}(A\wedge B)\},\mathbf{T}A,\mathbf{T}B}\mathbf{T}\wedge$. If no confusion arises we say that *a rule applies to* Γ or equivalently *a rule applies to H*. Finally with *Rule*(H) we mean the rule related to H (in our example *Rule*$(\mathbf{T}(A \wedge B))$ is $\mathbf{T}\wedge$).

For every proof table for S, the depth and the number of symbols occurring in the nodes is linearly bounded in the number of symbols occurring in S (see [AFM04] for further details). This is the key feature to implement a depth-first decision procedure whose space complexity is $O(n \lg n)$ (as a matter of fact, it is well known that to generate all the proof tables in the search space and to visit them with a depth-first strategy, it is sufficient to have a stack containing, for every node of the visited branch, the index of the main swff and a bit to store if the leftmost branch has been visited [Hud93]).

We emphasize that the sign $\mathbf{F_c}$ is introduced to give a specialized treatment of the negated formulas. In this sense the rules for the formulas signed with $\mathbf{F_c}$ could be rewritten replacing in the $\mathbf{F_c}$-rules every occurrence of the sign $\mathbf{F_c}$ with $\mathbf{T}\neg$.

Given a set of swffs S, the signed atoms of S are the elements of the set $\delta_S = \{H | H \in S$ and H is a signed atom$\}$. We say that S contains a complementary pair iff $\{\mathbf{T}A, \mathbf{F}A\} \subseteq S$ or $\{\mathbf{T}A, \mathbf{F_c}A\} \subseteq S$. Given δ_S, we denote with σ_{δ_S} the (classical) model defined as follows: if $\mathbf{T}p \in \delta_S$, then $\sigma_{\delta_S}(p) = true$, $\sigma_{\delta_S}(p) = false$ otherwise.

Given a classical model σ, (I) $\sigma \rhd \mathbf{T}H$ iff $\sigma \models H$; (II) $\sigma \rhd \mathbf{F}H$ and $\sigma \rhd \mathbf{F_c}H$ iff $\sigma \not\models H$. Given a set of swffs S, a swff H (respectively a set of swffs S') and the set of signed atoms δ_S defined as above, we say that δ_S *r*ealizes H (respectively δ_S *realizes* S'), and we write $\delta_S \rhd H$ (respectively $\delta_S \rhd S'$), iff there exists a classical model σ fulfilling the following conditions:

(i) $\sigma \rhd H$ (respectively $\sigma \rhd H'$ for every $H' \in S'$).

(ii) if $\mathbf{T}p \in \delta_S$, then $\sigma(p) = true$;

(iii) if $\mathbf{F}p \in \delta_S$ or $\mathbf{F_c}p \in \delta_S$, then $\sigma(p) = false$

In other words, the relation $\rhd$ between δ_S and a signed formula H holds if there exists a classical model σ both realizing δ_S and H. If $\{\mathbf{T}p, \mathbf{F_c}p, \mathbf{F}p\} \cap S = \emptyset$ then there is no condition on the truth value that σ gives to p.

$$\frac{S, \mathbf{T}(A \wedge B)}{S, \mathbf{T}A, \mathbf{T}B}\mathbf{T}\wedge \quad \frac{S, \mathbf{F}(A \wedge B)}{S, \mathbf{F}A | S, \mathbf{F}B}\mathbf{F}\wedge \quad \frac{S, \mathbf{F_c}(A \wedge B)}{S_c, \mathbf{F_c}A | S_c, \mathbf{F_c}B}\mathbf{F_c}\wedge$$

$$\frac{S, \mathbf{T}(A \vee B)}{S, \mathbf{T}A | S, \mathbf{T}B}\mathbf{T}\vee \quad \frac{S, \mathbf{F}(A \vee B)}{S, \mathbf{F}A, \mathbf{F}B}\mathbf{F}\vee \quad \frac{S, \mathbf{F_c}(A \vee B)}{S, \mathbf{F_c}A, \mathbf{F_c}B}\mathbf{F_c}\vee$$

$$\frac{S, \mathbf{T}A, \mathbf{T}(A \rightarrow B)}{S, \mathbf{T}A, \mathbf{T}B}\mathbf{T} \rightarrow Atom, \text{ with } A \text{ an atom}$$

$$\frac{S, \mathbf{F}(A \rightarrow B)}{S_c, \mathbf{T}A, \mathbf{F}B}\mathbf{F} \rightarrow \quad \frac{S, \mathbf{F_c}(A \rightarrow B)}{S_c, \mathbf{T}A, \mathbf{F_c}B}\mathbf{F_c} \rightarrow$$

$$\frac{S, \mathbf{T}(\neg A)}{S, \mathbf{F_c}A}\mathbf{T}\neg \quad \frac{S, \mathbf{F}(\neg A)}{S_c, \mathbf{T}A}\mathbf{F}\neg \quad \frac{S, \mathbf{F_c}(\neg A)}{S_c, \mathbf{T}A}\mathbf{F_c}\neg$$

$$\frac{S, \mathbf{T}((A \wedge B) \rightarrow C)}{S, \mathbf{T}(A \rightarrow (B \rightarrow C))}\mathbf{T} \rightarrow \wedge \quad \frac{S, \mathbf{T}(\neg A \rightarrow B)}{S_c, \mathbf{T}A | S, \mathbf{T}B}\mathbf{T} \rightarrow \neg$$

$$\frac{S, \mathbf{T}((A \vee B) \rightarrow C)}{S, \mathbf{T}(A \rightarrow p), \mathbf{T}(B \rightarrow p), \mathbf{T}(p \rightarrow C)}\mathbf{T} \rightarrow \vee$$

$$\frac{S, \mathbf{T}((A \rightarrow B) \rightarrow C)}{S_c, \mathbf{T}A, \mathbf{F}p, \mathbf{T}(p \rightarrow C), \mathbf{T}(B \rightarrow p) | S, \mathbf{T}C}\mathbf{T} \rightarrow\rightarrow$$

where $S_c = \{\mathbf{T}A | \mathbf{T}A \in S\} \cup \{\mathbf{F_c}A | \mathbf{F_c}A \in S\}$ and
p is a new atom

Table 1: the **Tab** calculus

We say that a wff or a swff is classically evaluable (*cle*-wff and *cle*-swff for short) iff conjunctions and disjunctions are the only connectives occurring in it. Finally, a set S of swffs is contradictory if at least one of the following conditions holds:

1. S contains a complementary pair;
2. S contains a *cle*-swff H such that $\delta_S \not\triangleright H$;
3. $\delta_S \not\triangleright S$ and for every propositional variable p occurring in S, $\mathbf{T}p \in S$ or $\mathbf{F_c}p \in S$.

Proposition 2.1 *If a set of swffs S is contradictory, then for every Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$, $\rho \not\triangleright S$.*

Proof: If S is contradictory because the first condition holds, then for some formula A, $\{\mathbf{T}A, \mathbf{F}A\} \subseteq S$ or $\{\mathbf{T}A, \mathbf{F_c}A\} \subseteq S$ holds. By the meaning of the signs and the definition of the forcing relation in Kripke models, the claim immediately follows. If S is contradictory because the second condition holds, then $\delta_S \not\triangleright H$. Thus, there is no classical model realizing both the signed atoms of S (that is δ_S) and H. Since H

is a *cle*-swff, its classical and intuitionistic realizability coincide, thus no Kripke model realizes S. If S is contradictory because the third condition holds, then let us suppose there exists a Kripke model $\underline{K} = \langle P,\leq,\rho,\Vdash\rangle$ such that $\rho \rhd S$. Then for every $p \in \mathcal{PV}(S)$, $\rho \Vdash p$ or $\rho \Vdash \neg p$ and this means that every world in $\underline{K}$ forces the same propositional variables occurring in S, that is for every $\alpha, \beta \in P$ and for every $p \in \mathcal{PV}(S)$, $\alpha \Vdash p$ iff $\beta \Vdash p$. Let $\phi \in P$ be a maximal state in the poset $\langle P, \leq, \rho\rangle$. Since ϕ behaves as a classical model, by hypothesis, $\phi \not\rhd S$. Then, since every world of $\underline{K}$ forces the same propositional variables of S we deduce that $\rho \not\rhd S$. □

A closed proof table is a proof table whose leaves are all contradictory sets. A closed proof table is a proof of the calculus: a formula A is provable if there exists a closed proof table for $\{\mathbf{F}A\}$. For every rule of the calculus it is easy to prove that if there exists a Kripke model $\underline{K} = \langle P,\leq,\rho,\Vdash\rangle$ and $\alpha \in P$ such that α realizes the premise of the rule, then there exists (a possibly different) Kripke model $\underline{K'} = \langle P', \leq', \rho', \Vdash'\rangle$ and $\beta \in P'$ such that β realizes the conclusion. This is the main step to prove the soundness of the calculus:

Theorem 2.2 (Soundness) *Let A be a wff. If there exists a closed proof table starting from $\{\mathbf{F}A\}$, then A is valid.*

3 The optimizations and the proof search algorithm

Below we describe a recursive procedure $\textsc{Tab}(S)$ which given a set S of swffs, returns either a closed proof table for S or `NULL` (if there exists a Kripke model realizing S).

To describe $\textsc{Tab}$ we use the following notation. Let S be a set of swffs, let $H \in S$ and let S_1 or $S_1 \mid S_2$ be the nodes of the proof tree obtained by applying to S the rule $\mathit{Rule}(H)$ corresponding to H. If Tab_1 and Tab_2 are closed proof tables for S_1 and S_2 respectively, then $\dfrac{S}{Tab_1}\mathit{Rule}(H)$ or $\dfrac{S}{Tab_1 \mid Tab_2}\mathit{Rule}(H)$ denote the closed proof table for S defined in the obvious way. Moreover, $\mathcal{R}_i(H)$ $(i = 1, 2)$ denotes the set containing the swffs of S_i which replaces H. For instance:
$\mathcal{R}_1(\mathbf{T}(A \wedge B)) = \{\,\mathbf{T}A, \mathbf{T}B\,\}$,
$\mathcal{R}_1(\mathbf{T}(A \vee B)) = \{\mathbf{T}A\}$, $\mathcal{R}_2(\mathbf{T}(A \vee B)) = \{\mathbf{T}B\}$,
$\mathcal{R}_1(\mathbf{T}((A \to B) \to C)) = \{\,\mathbf{T}A, \mathbf{F}p, \mathbf{T}(B \to p), \mathbf{T}(p \to C)\,\}$,
$\mathcal{R}_2(\mathbf{T}((A \to B) \to C)) = \{\mathbf{T}C\}$.

As stated in the introduction, $\textsc{Tab}$ uses substitutions. For our purposes, the only substitutions we are interested in are permutations from $\mathcal{PV}$ to $\mathcal{PV}$. Given a substitution τ and a swff H (respectively, a set of swffs S and tableau T) $\tau(H)$ (respectively, $\tau(S)$ and $\tau(T)$) means the swff (respectively, the set of swffs and the tableau) obtained by applying the substitution in the obvious way.

The procedure $\textsc{Tab}$ divides the formulas in six groups according to their behavior with respect to branching and backtracking:
$\mathcal{C}_1 = \{\mathbf{T}(A \wedge B)$, $\mathbf{F}(A \vee B)$, $\mathbf{F_c}(A \vee B)$, $\mathbf{T}(\neg A)$, $\mathbf{T}(p \to A)$ with p an atom, $\mathbf{T}((A \wedge B) \to C)$, $\mathbf{T}((A \vee B) \to C)\}$;
$\mathcal{C}_2 = \{\mathbf{T}(A \vee B)$, $\mathbf{F}(A \wedge B)$;
$\mathcal{C}_3 = \{\mathbf{F}(\neg A)$, $\mathbf{F}(A \to B)\}$;

$\mathcal{C}_4 = \{\mathbf{T}((A \to B) \to C),\ \mathbf{T}(\neg A \to B)\}$;
$\mathcal{C}_5 = \{\mathbf{F_c}(A \to B),\ \mathbf{F_c}(\neg A)\}$;
$\mathcal{C}_6 = \{\mathbf{F_c}(A \wedge B)\}$.
We call $\mathcal{C}_i$-swffs ($i = 1, \ldots, 6$), respectively $\mathcal{C}_i$-rules, the swffs of the group $\mathcal{C}_i$, respectively the rules related to $\mathcal{C}_i$-swffs.

The intuition behind these groups can be explained as follows. It is well known that in classical logic the order in which the rules are applied does not affect the completeness of the decision procedure: if a proof exists it is found independently of the order in which the rules are applied. Thus the search space consists of a single proof tree whose branches have to be visited by the decision procedure. The rules having two conclusions give rise to branches. Rules of this kind are $\mathbf{T}\vee$ and $\mathbf{F}\wedge$. In intuitionistic logic the order in which the rules are applied is relevant and affect the completeness. Given a set Γ, there are many ways to go on with the proof (that is many swffs can be chosen as main swff). Since the order is relevant, if the choice of a swff as main swff does not give a closed proof table, we have to backtrack and try with another swff as main swff. This means that in intuitionistic logic there is a space of proof tables to be visited by backtracking. Rules requiring backtracking are, i.e., $\mathbf{F} \to$, $\mathbf{T} \to\to$. In order to bound time consumption, Tab applies the rules not requiring branching and backtracking first, then the rules not requiring backtracking, finally the rules requiring backtracking. In the Completeness Lemma (Lemma 4.1, page 11) we prove that we do not lose completeness if $\mathcal{C}_5$ and $\mathcal{C}_6$-rules are applied only when no other rule applies. Thus the application of $\mathcal{C}_5$ and $\mathcal{C}_6$-rules is invertible and no backtracking is required. On the other hand, to get completeness, backtracking is unavoidable when $\mathcal{C}_3$ and $\mathcal{C}_4$-rules are applied.

Now we come to the optimizations. First we discuss two checks that allow us to bound the depth of the proofs. Let S be a set such that $\sigma_{\delta_S} \rhd S$. Thus the Kripke model coinciding with the classical model σ_{δ_S} realizes S and we do not need to go on with the proof. The second check is related to Point 3 in the definition of contradictory set. If $\delta_S \not\rhd S$ and every propositional variable occurring in S occurs in S as swff signed $\mathbf{T}$ or $\mathbf{F_c}$, then there is no Kripke model realizing S and we do not need to proceed with the proof. Although these checks could be performed after every rule application, our strategy performs it when neither a $\mathcal{C}_1$ nor a $\mathcal{C}_2$-rule applies to S (in Section 5 this optimization is referred as **opt1**).

In order to bound branching, Tab treats in a particular way the *cle*-swffs in S, that is the swffs in which only $\wedge$ and $\vee$ occur (in other words swffs whose intuitionistic truth coincides with classical truth). When Tab treats $\mathcal{C}_2$-swffs (Point 3 of the algorithm given below), first of all Tab checks if in S there exists a *cle*-swff H such that $\delta_S \not\rhd H$. If S fulfills this condition, then S is not realizable, as we pointed out in Proposition 2.1. Otherwise, if in S a *c*le-swff H occurs, such that σ_{δ_S} does not satisfy H, the splitting rule $Rule(H)$ is applied (we recall that σ_{δ_S} is the classical model defined from S by taking as true the atoms p such that $\mathbf{T}p \in S$). Thus the *c*le-swffs of S, satisfying the underlying model, are never applied. Despite this, from the realizability of one among $(S \setminus \{H\}) \cup \mathcal{R}_1(H)$ and $(S \setminus \{H\}) \cup \mathcal{R}_2(H)$ we have enough information to prove the realizability of S. In other words, we consider only the *cle*-swffs of $\mathcal{C}_2$ that are not realized by the model underlying S (see Point 3 and Completeness Lemma for the details). As an example consider

$$\begin{aligned} S \ = \{ \ & \mathbf{F}(P0 \wedge P2), \mathbf{F}(P0 \wedge P4), \mathbf{F}(P2 \wedge P4), \mathbf{F}(P1 \wedge P3), \mathbf{F}(P1 \wedge P5), \\ & \mathbf{F}(P3 \wedge P5), \mathbf{T}(P0 \vee P1), \mathbf{T}(P2 \vee P3), \mathbf{T}(P4 \vee P5) \ \}. \end{aligned}$$

Since σ_{δ_S} realizes the **F**-swffs in S but σ_{δ_S} does not realize any of the **T**-swffs of S, then Tab chooses one of them, let us suppose $H = \mathbf{T}(P0 \vee P1)$. The rule $\mathbf{T}\vee$ is applied to S and $S_1 = (S \setminus \{H\}) \cup \{\mathbf{T}P0\}$ and $S_2 = (S \setminus \{H\}) \cup \{\mathbf{T}P1\}$ are the subsequent sets of S. Now consider S_1. Since $\sigma_{\delta_{S_1}}$ realizes $\mathbf{T}P0$ and all the **F**-swffs in S_1, but $\sigma_{\delta_{S_1}}$ realizes neither $\mathbf{T}(P2 \vee P3)$ nor $\mathbf{T}(P4 \vee P5)$, Tab chooses one of them, let us suppose $H = \mathbf{T}(P2 \vee P3)$. The rule $\mathbf{T}\vee$ is applied to S_1 and $S_3 = (S_1 \setminus \{H\}) \cup \{\mathbf{T}P2\}$ and $S_4 = (S_1 \setminus \{H\}) \cup \{\mathbf{T}P3\}$ are the subsequent sets of S_1. Since δ_{S_3} does not realize $\mathbf{F}(P0 \wedge P2)$ we deduce that S_3 is contradictory. Similarly for the other sets. We emphasize that at every step, the $\mathcal{C}_2$-rules applicable to S_i are only the ones where the related swffs are not realized by $\sigma_{\delta_{S_i}}$. Without this strategy a huge closed proof table could arise (in Section 5 this optimization is referred as **opt2**).

To bound the search space, [Wei98] describes a decision procedure in which backtracking is bounded by a semantical technique inspired by the completeness theorem. The completeness theorem proves the satisfiability (realizability in our context) of a set S under the hypothesis that S does not have any closed proof table. As an example, let $S = \{\mathbf{F}(A \to B), \mathbf{T}((A \to B) \to C), \mathbf{F}C, \mathbf{T}D\}$. From S we can define the Kripke model $\underline{K}(S) = \langle P, \leq, \rho, \Vdash \rangle$ such that $P = \{\rho\}$ and $\rho \Vdash D$. Note that $\underline{K}(S)$ realizes $\mathbf{T}D$ but $\underline{K}(S)$ does not realize S. To prove the realizability of S, the realizability of $S_1 = \{\mathbf{T}A, \mathbf{F}B, \mathbf{T}((A \to B) \to C), \mathbf{T}D\}$ and one between $S_2 = \{\mathbf{T}A, \mathbf{F}p, \mathbf{T}(B \to p), \mathbf{T}(p \to C), \mathbf{T}D\}$ and $S_3 = \{\mathbf{F}(A \to B), \mathbf{T}C, \mathbf{F}C, \mathbf{T}D\}$ have to be proved. Since S_3 is not realizable, the realizability of S_1 and S_2 must be proved. From S_1 we define the Kripke model $\underline{K}(S_1) = \langle \{\alpha\}, \leq, \alpha, \Vdash \rangle$, where $\alpha \leq \alpha$, $\alpha \Vdash D$ and $\alpha \Vdash A$ such that $\underline{K}(S_1) \rhd S_1$. If we glue $\underline{K}(S1)$ above $\underline{K}(S)$ we get a new Kripke model $\underline{K} = \langle \{\rho, \alpha\}, \leq, \rho, \Vdash \rangle$ where $\rho \leq \rho$, $\rho \leq \alpha$, $\alpha \leq \alpha$, $\rho \Vdash D$,$\alpha \Vdash D$ and $\alpha \Vdash A$. Since $\underline{K} \rhd S$, we do not need to apply $\mathbf{T} \to\to$ to S in order to obtain $S_2 = \{\mathbf{T}A, \mathbf{F}p, \mathbf{T}(B \to p), \mathbf{T}(B \to C), \mathbf{T}D\}$ (from S_2 a Kripke model $\underline{K}(S_2)$ is definable; $\underline{K}(S_2)$ glued above $\underline{K}(S)$ gives rise to a Kripke model realizing S). In this case the work on S_2 is spared. In the general case, see [Wei98], the information collected from non closed proof tables built from a set S is used to build a Kripke model $\underline{K}$. As a matter of fact, let S be a set such that no $\mathcal{C}_1$ or $\mathcal{C}_2$-rule is applicable. Let $\{H_1, \ldots, H_u\} \subseteq S$ the $\mathcal{C}_3$ and $\mathcal{C}_4$-swffs of S. If there exists a H_j such that $\underline{K} \not\rhd H_j$, then $Rule(H_j)$ have to be applied. If a closed proof table is found, then S is not realizable, otherwise the Kripke model $\underline{K}$ can be extended in a new one, $\underline{K}_j$ satisfying H_j. The procedure of [Wei98] continues until a closed proof table or a Kripke model $\underline{K}_i$, $1 \leq i \leq u$, such that $\underline{K}_i \rhd \{H_1, \ldots, H_u\}$ is found. The procedure prunes the search space, since in S not all the swffs requiring backtracking are considered, but only the swffs which, when checked, are not realized from the Kripke model at hand.

Now, consider $S = \{\mathbf{F}(A \to B), \mathbf{F}(C \to D)\}$. From S we can define the Kripke model $\underline{K}(S) = \langle P, \leq, \rho, \Vdash \rangle$ such that $P = \{\rho\}$ and $\Vdash$ is the empty set. $\underline{K}(S)$ does not realize S. By applying $\mathbf{F} \to$ to S with $\mathbf{F}(A \to B)$ as the main formula we get $S_1 = \{\mathbf{T}A, \mathbf{F}B\}$. The underlying model is $\underline{K}(S_1) = \langle \{\alpha\}, \leq, \alpha, \Vdash \rangle$ with $\alpha \Vdash A$. $\underline{K}(S_1)$ glued above $\underline{K}(S)$ gives rise to a model that does not realize $\mathbf{F}(C \to D)$. Thus we must backtrack. We apply $\mathbf{F} \to$ to S with $\mathbf{F}(C \to D)$ as the main formula. We get $S_2 = \{\mathbf{T}C, \mathbf{F}D\}$. The underlying model is $\underline{K}(S_2) = \langle\{\beta\},\leq,\beta,\Vdash\rangle$ such that $\beta \leq \beta$ and $\beta \Vdash C$ realizes S_2. By gluing $\underline{K}(S_1)$ and $\underline{K}(S_2)$ above $\underline{K}(S)$ the resulting model $\underline{K} = \langle\{\rho, \alpha, \beta\},\leq,\rho,\Vdash\rangle$ such that

$$\rho \leq \alpha, \rho \leq \beta, \rho \leq \rho, \alpha \leq \alpha, \beta \leq \beta, \alpha \Vdash A \text{ and } \beta \Vdash C$$

realizes S. But by a permutation $\tau : \mathcal{PV} \to \mathcal{PV}$ such that $\tau(C) = A$ and $\tau(D) = B$, $\tau(S_2) = S_1$ and we can build $\underline{K}(S_2) = \langle P, \leq, \Vdash', \rho, \rangle$ from $\underline{K}(S_1)$ as follows: $\underline{K}(S_2)$ has the same poset as $\underline{K}(S_1)$ and $\Vdash'$ is: for every $\alpha \in P$ and for every $p \in \mathcal{PV}$, $\alpha \Vdash' p$ iff $\alpha \Vdash \tau(p)$. In other words, $\underline{K}(S_1)$ can be translated into $\underline{K}(S_2)$ via τ and we can avoid backtracking on S. As another example consider

$$\begin{aligned} S \;=\{\;\; & \mathbf{T}(((P0 \to (P1 \vee P2)) \to (P1 \vee P2))), \\ & \mathbf{T}(((P2 \to (P1 \vee P0)) \to (P1 \vee P0))), \\ & \mathbf{T}(((P1 \to (P2 \vee P0)) \to (P2 \vee P0))), \mathbf{F}((P1 \vee (P2 \vee P0)))\ \}, \end{aligned}$$

where only a few steps are needed to obtain S starting from $\{\mathbf{F}H\}$, where H is the axiom schema $\bigwedge_{i=0}^{2} \Big((P_i \to \bigvee_{j\neq i} P_j) \to \bigvee_{j\neq i} P_j\Big) \to \bigvee_{i=0}^{2} P_i$ characterizing the logic of binary trees (a logic in the family of k-ary trees logics, [CZ97], also known as Gabbay-de Jongh logics). From S we can define the model $\underline{K}(S) = \langle P,\leq,\rho,\Vdash\rangle$ such that $P = \{\rho\}$ and $\Vdash$ is the empty set. $\underline{K}(S)$ does not realize S. By applying $\mathbf{T} \to\to$ to S with $H = \mathbf{T}(((P0 \to (P1 \vee P2)) \to (P1 \vee P2)))$ we get $S_1 = (S \setminus \{H\}) \cup \mathcal{R}_2(H)$ and $S_2 = (S \setminus \{H\})_c \cup \mathcal{R}_1(H)$. Since S_1 is not realizable, to prove the realizability of S we have to prove the realizability of S_2. S_2 defines the Kripke model $\underline{K}(S_2) = \langle\{\alpha\},\leq,\alpha,\Vdash\rangle$, where $\alpha \leq \alpha$ and $\alpha \Vdash P0$. Thus $\underline{K}(S_2) \rhd S_2$ holds. Now, if we glue $\underline{K}(S_2)$ above $\underline{K}(S)$ we get a new model $\underline{K}'(S) = \langle\{\rho, \alpha\},\leq,\rho,\Vdash\rangle$, where $\rho \leq \rho, \rho \leq \alpha, \alpha \leq \alpha$ and $\alpha \Vdash P0$. $\underline{K}'(S)$ does not realize S. Thus we must backtrack twice:

(i) by applying $\mathbf{T} \to\to$ to S with $H = \mathbf{T}(((P2 \to (P1 \vee P0)) \to (P1 \vee P0)))$ we get, $S_3 = (S \setminus \{H\})_c \cup \mathcal{R}_1$ and $S_4 = (S \setminus \{H\}) \cup \mathcal{R}_2(H)$;

(ii) by applying $\mathbf{T} \to\to$ to S with $H = \mathbf{T}(((P1 \to (P2 \vee P0)) \to (P2 \vee P0)))$ we get $S_5 = (S \setminus \{H\}) \cup \mathcal{R}_2(H)$ and $S_6 = (S \setminus \{H\})_c \cup \mathcal{R}_1(H)$.

In a few steps we find that S_4 and S_6 are not realizable. From S_3 we define the Kripke model $\underline{K}(S_3) = \langle\{\beta\},\leq,\beta,\Vdash\rangle$ where $\beta \leq \beta$ and $\beta \Vdash P2$. $\underline{K}(S_3) \rhd S_3$. From S_5 we define the Kripke model $\underline{K}(S_5) = \langle\{\gamma\},\leq,\gamma,\Vdash\rangle$ where $\gamma \leq \gamma$ and $\gamma \Vdash P1$. $\underline{K}(S_5) \rhd S_5$. Thus by gluing $\underline{K}(S_2), \underline{K}(S_3)$ and $\underline{K}(S_5)$ above $\underline{K}(S)$ we get a model $\underline{K}$ realizing S. Since we can define the permutations τ_1 and τ_2 such that $\tau_1(S_3) = S_2$ and $\tau_2(S_5) = S_2$ we can avoid backtracking. Thus no proof of realizability of S_3 or S_5 is needed and the Kripke models realizing S_3 and S_5 can be obtained by applying the permutations on the forcing relation of the Kripke model for S_2.

Thus to avoid backtracking Tab builds a permutation τ between sets of swffs. Let H be $\mathcal{C}_3$-swff. Before applying $Rule(H)$ we check if there exists a permutation τ from $\mathcal{PV}(S)$ to $\mathcal{PV}(S)$ such that $\tau((S \setminus \{H\})_c \cup \mathcal{R}_1(H)) = (S \setminus \{H'\})_c \cup \mathcal{R}_1(H')$. We already know that the set $(S \setminus \{H'\})_c \cup \mathcal{R}_1(H')$ obtained treating H' is realizable by a Kripke model $\underline{K}'$. Since we have a permutation τ, then the set $(S \setminus \{H\})_c \cup \mathcal{R}_1(H)$ is realized by the Kripke model $\underline{K}$ having the same poset as $\underline{K}'$ and such that for every world α and every propositional variable p, $\alpha \Vdash_{\underline{K}'} p$ iff $\alpha \Vdash_{\underline{K}} \tau(p)$. This means that the permutation τ allows us to go from $\underline{K}'$ to $\underline{K}$ (and the permutation τ^{-1} allows us to go from $\underline{K}$ to $\underline{K}'$). In particular, τ and τ^{-1} translate the forcing relation between the models. Analogously if H is a $\mathcal{C}_4$-swff. We emphasize that given a Kripke model $\underline{K}$, a permutation τ and a swff H, $\underline{K} \rhd H$ does not imply $\underline{K} \rhd \tau(H)$. Thus we have taken a different route with respect to [Wei98], where the realizability of $\tau(H)$ is checked on $\underline{K}$ that realizes H and the two methods work in different situations. We emphasize that both methods imply a certain computational cost. The method of [Wei98] implies checking the realizability on a Kripke model, which is time consuming for swffs of the kind $\mathbf{T}(A \to B)$. Our method can be time consuming if we perform a search of a permutation among the $Pv(S)!$ possible permutations. However, as we describe in Section 5, the procedure searches in a small subset of all possible permutations (in Section 5, this optimization is referred as **opt3**).

Finally, we could also define a permutation to prove that a set is not realizable. As a matter of fact, if S is not realizable and there exists a permutation τ such that $\tau(S) = S'$, then S' is not realizable. Thus, given a set S and a $\mathcal{C}_2$ or $\mathcal{C}_6$-swff $H \in S$, if $(S \setminus \{H\}) \cup \mathcal{R}_1(H)$ is closed and there exists a permutation τ such that $\tau((S \setminus \{H\}) \cup \mathcal{R}_1(H)) = (S \setminus \{H\}) \cup \mathcal{R}_2(H)$ then $(S \setminus \{H\}) \cup \mathcal{R}_2(H)$ is not realizable and the tableau proof for $(S \setminus \{H\}) \cup \mathcal{R}_1(H)$ can be translated via τ in a tableau proof for $(S \setminus \{H\}) \cup \mathcal{R}_2(H)$ (see Points 3 and 6 of Tab). As a trivial application, consider a valid wff $H(\underline{p})$, where $\underline{p} = \{p_1, \ldots, p_n\}$ are all the propositional variables occurring in H. To prove that $\{\mathbf{F}(H(\underline{p}) \wedge H(\underline{q}))\}$ is closed, it is sufficient to prove, by an application of $\mathbf{F}\wedge$, that $\{\mathbf{F}H(\underline{p})\}$ is closed and there exists a permutation such that $\{\mathbf{F}H(\underline{p})\} = \tau(\{\mathbf{F}H(\underline{q})\})$.

To save work space, we describe Tab in natural language. The algorithm is divided in seven main points. We recall that the input of **Tab** is a set S of swffs. If S is realizable, then **Tab** returns `NULL`, otherwise **Tab** returns a closed proof table for S. In the following description, given a set V of swffs, Tab (V) is the recursive call to Tab with actual parameter V. Some instructions 'return `NULL`' are labeled with **r1**,..., **r6**. In the Completeness Lemma we refer to such instructions by means of these labels.
Function Tab (S)
1. If S contains a complementary pair, then Tab returns the proof S;

2. If a $\mathcal{C}_1$-rule applies to S, then let H be a $\mathcal{C}_1$-swff. If Tab$((S \setminus \{H\}) \cup \mathcal{R}_1(H))$ returns a proof π, then Tab returns the proof $\dfrac{S}{\pi}\mathit{Rule}(H)$, otherwise Tab returns `NULL` (**r1**);

3. If a $\mathcal{C}_2$-rule applies to S, then if there exists a $\mathcal{C}_2$-swff H such that H is a *cle*-swff and $\delta_S \not\rhd H$, then Tab returns (the proof) S. Otherwise, let H be a *cle*-swff such that $\sigma_{\delta_S} \not\rhd H$, if there is any, otherwise let H be a $\mathcal{C}_2$-swff. Let $\pi_1 = $ Tab$(S \setminus \{H\} \cup \mathcal{R}_1(H))$. If π_1 is `NULL`, then Tab returns `NULL`. If there exists a permutation τ such that $(S \setminus \{H\}) \cup \mathcal{R}_1(H) = \tau(S \setminus \{H\}) \cup \mathcal{R}_2(H))$, then Tab returns the proof $\dfrac{S}{\pi_1 \mid \tau^{-1}(\pi_1)}\mathit{Rule}(H)$. If such a permutation does not exist, then let $\pi_2 = $ Tab$(S \setminus \{H\} \cup \mathcal{R}_2(H))$. If π_2 is a proof, then Tab returns the proof $\dfrac{S}{\pi_1 \mid \pi_2}\mathit{Rule}(H)$, otherwise ($\pi_2$ is `NULL`) Tab returns `NULL`;

4. If a $\mathcal{C}_3$ or $\mathcal{C}_4$-rule applies to S, then Tab proceeds as follows: if $\sigma_{\delta_S} \rhd S$, then Tab returns `NULL` (**r2**). If for every $p \in \mathcal{PV}(S)$, $\mathbf{T}p \in S$ or $\mathbf{F_c}p \in S$, then TAB returns S. If the previous points do not apply, then TAB carries out the following Points **4.1** and **4.2**:

4.1 Let $\{H_1, \dots H_n\}$ be all the $\mathcal{C}_3$-swffs in S. For $i = 1, \dots, n$, the following instructions are iterated: if there is no swff H_j ($j \in \{1, \dots, i-1\}$) and a permutation τ such that $(S \setminus \{H_j\})_c \cup \mathcal{R}_1(H_j) = \tau((S \setminus \{H_i\})_c \cup \mathcal{R}_1(H_i))$, then let $\pi = \text{TAB}((S \setminus \{H_i\})_c \cup \mathcal{R}_1(H_i))$. If π is a proof, then TAB returns the proof $\dfrac{S}{\pi}\mathit{Rule}(H_i)$;

4.2 Let $\{H_1, \dots H_n\}$ be all the $\mathcal{C}_4$-swffs in S. For $i = 1, \dots, n$, the following Points (**4.2.1**) and (**4.2.2**) are iterated.

(**4.2.1**) If there is neither swff H_j ($j \in \{1, \dots, i-1\}$) nor a permutation τ such that $(S \setminus \{H_j\}) \cup \mathcal{R}_2(H_j) = \tau((S \setminus \{H_i\}) \cup \mathcal{R}_2(H_i))$, then let $\pi_{2,i} = \text{TAB}((S \setminus \{H_i\}) \cup \mathcal{R}_2(H_i))$. If $\pi_{2,i}$ is `NULL`, then TAB returns `NULL` (**r3**). If there is neither swff H_j, with $j \in \{1, \dots, i-1\}$, nor a permutation τ such that $(S \setminus \{H_j\})_c \cup \mathcal{R}_1(H_j) = \tau((S \setminus \{H_i\})_c \cup \mathcal{R}_1(H_i))$, then if $\text{TAB}((S \setminus \{H_i\})_c \cup \mathcal{R}_1(H_i))$ returns a proof π_1, TAB returns the proof $\dfrac{S}{\pi_1 \mid \pi_{2,i}}\mathit{Rule}(H_i)$.

(**4.2.2**) If Point (**4.2.1**) does not hold, then there exists a permutation τ and a swff H_j ($j \in \{1, \dots, i-1\}$) such that $(S \setminus \{H_j\}) \cup \mathcal{R}_2(H_j) = \tau((S \setminus \{H_i\}) \cup \mathcal{R}_2(H_i))$. If there is no swff H_u, with $u \in \{1, \dots, i-1\}$, and a permutation τ such that $(S \setminus \{H_u\})_c \cup \mathcal{R}_1(H_u) = \tau((S \setminus \{H_i\})_c \cup \mathcal{R}_1(H_i))$, then if $\text{TAB}((S \setminus \{H_i\})_c \cup \mathcal{R}_1(H_i))$ returns a proof π_1, TAB returns the proof $\dfrac{S}{\pi_1 \mid \tau^{-1}(\pi_{2,j})}\mathit{Rule}(H_i)$.

If in Points (**4.1**) and (**4.2**) TAB does not find any closed proof table, then TAB returns `NULL` (**r4**).

5. If a $\mathcal{C}_5$-rule applies to S, then let H be a $\mathcal{C}_5$-swff. If $\text{TAB}((S \setminus \{H\})_c \cup \mathcal{R}_1(H))$ returns a proof π, then TAB returns the proof $\dfrac{S}{\pi}\mathit{Rule}(H)$, otherwise TAB returns `NULL`;

6. If a $\mathcal{C}_6$-rule applies to S, then let H be a $\mathcal{C}_6$-swff. Let $\pi_1 = \text{TAB}((S \setminus \{H\})_c \cup \mathcal{R}_1(H))$. If π_1 is `NULL`, then TAB returns `NULL`. If there exists a permutation τ such that $(S \setminus \{H\})_c \cup \mathcal{R}_1(H) = \tau((S \setminus \{H\})_c \cup \mathcal{R}_2(H))$, then TAB returns the proof $\dfrac{S}{\pi_1 \mid \tau^{-1}(\pi_1)}\mathit{Rule}(H)$. If such a permutation does not exist, then let $\pi_2 = \text{TAB}((S \setminus \{H\})_c \cup \mathcal{R}_2(H))$. If π_2 is a proof, then TAB returns $\dfrac{S}{\pi_1 \mid \pi_2}\mathit{Rule}(H)$, otherwise ($\pi_2$ is `NULL`) TAB returns `NULL` (**r5**);

7. If none of the previous points apply, then TAB returns `NULL` (**r6**).

END FUNCTION TAB.

Point 4 deserves some comments. If the classical model σ_{δ_S} realizes S (condition "$\sigma_{\delta_S} \rhd S$") then such a model is a Kripke model realizing S. If for every propositional variable $p \in Pv(S)$, $\mathbf{T}p \in S$ or $\mathbf{F_c}p \in S$ holds, then the subsequent sets of S do not contain more information than S and from $\sigma_{\delta_S} \not\rhd S$ we deduce $\delta_S \not\rhd S$. Since $\delta_S \not\rhd S$, then S is not realizable (see Proposition 2.1). The iteration in Points **4.1** and **4.2** can be summarized as follows: a proof for S is searched: for every $\mathcal{C}_3$ or $\mathcal{C}_4$-swff H in S, $\mathit{Rule}(H)$ is applied to S. If no proof is found, then S is realizable. Now consider $\mathcal{C}_3$-swffs. If for a previous iteration j the set obtained by applying $\mathit{Rule}(H_j)$ to S is realizable and can be translated via a permutation τ in $(S \setminus \{H_i\})_c \cup \mathcal{R}(H_i)$, then TAB does not apply $Rule(H_i)$ to S. The permutation τ and the realizability of $(S \setminus \{H_j\})_c \cup \mathcal{R}(H_j)$ imply the realizability of $(S \setminus \{H_i\})_c \cup \mathcal{R}(H_i)$ (see Case 4 in Completeness Lemma). Tab applies the same idea in Point **4.2** to $\mathcal{C}_4$-swffs of S. This point is more complex because $\mathcal{C}_4$-rules have two conclusions.

4 Completeness

In order to prove the completeness of Tab, we prove that given a set of swffs S, if the call Tab(S) returns `NULL`, then we have enough information to build a countermodel $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ such that $\rho \rhd S$. To prove the proposition we need to introduce the function deg defined as follows:

- if p is an atom, then $\deg(p) = 0$;
- $\deg(A \wedge B) = \deg(A) + \deg(B) + 2$;
- $\deg(A \vee B) = \deg(A) + \deg(B) + 3$;
- $\deg(A \rightarrow B) = \deg(A) + \deg(B) +$ (number of implications occurring in A) + 1;
- $\deg(\neg A) = \deg(A) + 1$;
- $\deg(S) = \sum_{H \in S} \deg(H)$.

It is easy to show that if S' is obtained from a set of swffs S by an application of a rule of **Tab**, then $\deg(S') < \deg(S)$.

Lemma 4.1 (Completeness) *Let S be a set of swffs and suppose that* Tab(S) *returns the* `NULL` *value. Then, there is a Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ such that $\rho \rhd S$.*

Proof: The proof goes by induction on the complexity of S, measured with respect to the function $\deg(S)$.
Basis: if $\deg(S) = 0$, then S contains atomic swffs only. Tab(S) carries out the instruction labeled (***r6***). Moreover, S does not contain sets of the kind $\{\mathbf{T}p, \mathbf{F}p\}$ and $\{\mathbf{T}p, \mathbf{F_c}p\}$. Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be the Kripke model such that $P = \{\rho\}$ and $\rho \Vdash p$ iff $\mathbf{T}p \in S$. It is easy to show that $\rho \rhd S$.

Step: Let us assume by induction hypothesis that the proposition holds for all sets S' such that $\deg(S') < \deg(S)$. We prove that the proposition holds for S by inspecting all the possible cases where the procedure returns the `NULL` value.
Case 1: the instruction labeled **r1** *has been performed.* By induction hypothesis there exists a Kripke model $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ such that $\rho \rhd (S \setminus \{H\}) \cup \mathcal{R}_1(H)$, with $H \in \mathcal{C}_1$. We prove $\rho \rhd H$ by proceeding according to the cases of H. If H is of the kind $\mathbf{T}(A \wedge B)$, then by induction hypothesis $\rho \rhd \{\mathbf{T}A, \mathbf{T}B\}$, thus $\rho \Vdash A$ and $\rho \Vdash B$, and therefore $\rho \Vdash A \wedge B$. This implies $\rho \rhd \mathbf{T}(A \wedge B)$. The other cases for $H \in \mathcal{C}_1$ are similar.
Case 2: the instruction labeled **r2** *has been performed.* Thus $\sigma_{\delta_S} \rhd S$ holds. We use σ_{δ_S} to define a Kripke model $\underline{K}$ with a single world ρ such that $\rho \Vdash p$ iff $\sigma(p) = true$. Since ρ behaves as a classical model, $\rho \rhd S$ holds.
Case 3: the instruction labeled **r3** *has been performed.* By induction hypothesis there

exists a model $\underline{K}$ such that $\rho \rhd (S \setminus \{H_i\}) \cup \mathcal{R}_2(H_i)$, where $H_i \in \mathcal{C}_4$. Let us suppose that H is of the kind $\mathbf{T}((A \to B) \to C)$, thus $\rho \rhd \mathbf{T}C$ and this implies $\rho \rhd H_i$. The proof goes similarly if H_i is of the kind $\mathbf{T}(\neg A \to B)$.

Case 4: the instruction labeled **r4** *has been performed.* This implies that: (i) for every $H \in S \cap \mathcal{C}_3$, we have two cases: (ia) $\textsc{Tab}((S \setminus \{H\})_c \cup \mathcal{R}_1(H)) = \texttt{NULL}$, thus by induction hypothesis there exists a Kripke model $\underline{K}_H = \langle P_H, \leq_H, \rho_H, \Vdash_H \rangle$ such that $\rho_H \rhd (S \setminus \{H\})_c \cup \mathcal{R}_1(H)$; (ib) there exists a permutation τ from $\mathcal{PV}(S)$ to $\mathcal{PV}(S)$ and a swff $H' \in S \cap \mathcal{C}_3$ such that $\textsc{Tab}((S \setminus \{H'\})_c \cup \mathcal{R}_1(H')) = \texttt{NULL}$ and $(S \setminus \{H'\})_c \cup \mathcal{R}_1(H') = \tau((S \setminus \{H\})_c \cup \mathcal{R}_1(H))$. Thus by Point (a) applied to H', there exists a Kripke model $\underline{K}_{H'} = \langle P_{H'}, \leq_{H'}, \rho_{H'}, \Vdash_{H'} \rangle$ such that $\rho_{H'} \rhd (S \setminus \{H'\})_c \cup \mathcal{R}_1(H')$. By using τ we can translate $\underline{K}_{H'}$ into a model $\underline{K}_H = \langle P_H, \leq_H, \rho_H, \Vdash_H \rangle$, where $P_H = P_{H'}$, $\leq_H = \leq_{H'}$, $\rho_H = \rho_{H'}$ and for every world $\alpha \in P_H$, if $p \in \mathcal{PV}(S)$, then $\alpha \Vdash_H \tau(p)$ iff $\alpha \Vdash_{H'} p$. By definition of $\underline{K}_H$, it follows $\underline{K}_H \rhd (S \setminus \{H\})_c \cup \mathcal{R}_1(H))$; (ii) for every $H \in S \cap \mathcal{C}_4$, we have two cases: (iia)$\textsc{Tab}((S \setminus \{H\})_c \cup \mathcal{R}_1(H)) = \texttt{NULL}$, thus by induction hypothesis there exists a Kripke model $\underline{K}_H = \langle P_H, \leq_H, \rho_H, \Vdash_H \rangle$ such that $\rho_H \rhd (S \setminus \{H\})_c \cup \mathcal{R}_1(H)$. (iib) there exist a permutation τ from $\mathcal{PV}(S)$ to $\mathcal{PV}(S)$ and a swff $H' \in S \cap \mathcal{C}_4$ such that $\textsc{Tab}((S \setminus \{H'\})_c \cup \mathcal{R}_1(H')) = \texttt{NULL}$ and $(S \setminus \{H'\})_c \cup \mathcal{R}_1(H') = \tau((S \setminus \{H\})_c \cup \mathcal{R}_1(H))$. Thus by Point (a) applied to H', there exists a Kripke model $\underline{K}_{H'} = \langle P_{H'}, \leq_{H'}, \rho_{H'}, \Vdash_{H'} \rangle$ such that $\rho_{H'} \rhd (S \setminus \{H'\})_c \cup \mathcal{R}_1(H')$. By using τ we can translate $\underline{K}_{H'}$ into a model $\underline{K}_H = \langle P_H, \leq_H, \rho_H, \Vdash_H \rangle$, where $P_H = P_{H'}$, $\leq_H = \leq_{H'}$, $\rho_H = \rho_{H'}$ and for every world $\alpha \in P_H$, if $p \in \mathcal{PV}(S)$, then $\alpha \Vdash_H \tau(p)$ iff $\alpha \Vdash_{H'} p$. By definition of $\underline{K}_H$, it follows $\underline{K}_H \rhd (S \setminus \{H\})_c \cup \mathcal{R}_1(H))$. Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke model defined as follows:

$$P = \bigcup_{H \in S \cap (\mathcal{C}_3 \bigcup \mathcal{C}_4)} P_H \cup \{\rho\};$$

$$\leq = \bigcup_{H \in S \cap (\mathcal{C}_3 \bigcup \mathcal{C}_4)} \leq_H \cup \{(\rho, \alpha) | \alpha \in P\};$$

$$\Vdash \ = \ \bigcup_{H \in S \cap (\mathcal{C}_3 \bigcup \mathcal{C}_4)} \Vdash_H \cup \{(\rho, p) | \mathbf{T}p \in S\}.$$

By construction of $\underline{K}$, $\rho \rhd S$

Case 5: the instruction labeled **r5** *has been performed.* We point out that $S \cap \mathcal{C}_1 = S \cap \mathcal{C}_2 = S \cap \mathcal{C}_3 = S \cap \mathcal{C}_4 = S \cap \mathcal{C}_5 = \emptyset$. By induction hypothesis there exists a model $\underline{K}_H = \langle P_H, \leq_H, \rho_H, \Vdash_H \rangle$ such that $\rho_H \rhd (S \setminus \{H\})_c \cup \mathcal{R}_2(H)$, where $H \in \mathcal{C}_6$. Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be a Kripke defined as follows: $P = P_H \cup \{\rho\}$, $\leq = \leq_H \cup \{(\rho, \alpha) | \alpha \in P\}$, $\Vdash = \Vdash_H \cup \{(\rho, p) | \mathbf{T}p \in S\}$. By the construction of $\underline{K}$, $\rho \rhd S$, in particular, by induction hypothesis $\rho_H \Vdash \neg B$ and therefore $\rho_H \Vdash \neg(A \wedge B)$. This implies $\rho \rhd \mathbf{F_c}(A \wedge B)$.

Case 6: the instruction **r6** *has been carried out.* In this case S contains atomic swffs and swffs of the kind $\mathbf{T}(p \to A)$ and with $\mathbf{T}p \notin S$. Let $\underline{K} = \langle P, \leq, \rho, \Vdash \rangle$ be the Kripke model such that $P = \{\rho\}$ and $\rho \Vdash p$ iff $\mathbf{T}p \in S$. It is easy to show that $\rho \rhd S$. As a matter of fact, if $\mathbf{T}(p \to A) \in S$, since $\mathbf{T}p \notin S$, $\rho \nVdash p$ therefore $\rho \Vdash p \to A$. □

By Lemma 4.1 we immediately get the completeness of $\textsc{Tab}$.

Theorem 4.2 (Completeness) *If $A \in \mathbf{Int}$, then* $\textsc{Tab}(\{\mathbf{F}A\})$ *returns a closed proof table starting from* $\mathbf{F}A$*.*

5 Implementation and Results

We have implemented Tab as an iterative procedure in C++ language. At present there are some features that are missing. First, there is no any kind of lexical normalization. This feature, together with backjumping ([Bak95]) and BCP ([Fre95]), only to give a partial list of the possible optimization techniques, is typical in theorem provers and will be one of the changes in the new version of the implementation. Moreover, when PITP applies $\mathcal{C}_3$ and $\mathcal{C}_4$-rules, the search for a permutation proceeds as follows: let S be a set of swffs and let H and H' be $\mathcal{C}_3$-swffs in S. PITP does not perform a full search among the $Pv(S)!$ possible permutations. PITP tries to build a permutation τ such that $H = \tau(H')$ and $\tau = \tau^{-1}$. If such a τ fulfills $(S \setminus \{H\})_c \cup \mathcal{R}_1(H) = \tau((S \setminus \{H'\})_c \cup \mathcal{R}_1(H'))$, then τ is used. Otherwise PITP considers that no permutation exists and solves $(S \setminus \{H'\})_c \cup \mathcal{R}_1(H')$. Analogously for $\mathcal{C}_4$-swffs. Since our search is narrow, many permutations are disregarded. This problem is made worse by the fact that conjunctions and disjunctions are not implemented as lists of formulas. Thus at present this optimization is applied only to two families of formulas of ILTP v1.1.1 library. Finally PITP does not implement the search for a permutation in Points 3 and 6 of Tab. Despite the fact that some optimizations are

	ft Prolog	ft C	LJT	STRIP	PITP
solved	188	199	175	202	215
(%)	68.6	72.6	63.9	73.7	78.5
proved	104	106	108	119	128
refuted	84	93	67	83	87
solved after:					
0-1s	173	185	166	178	190
1-10s	5	6	4	11	10
10-100s	6	7	2	11	9
100-600s	4	1	3	2	6
(>600s)	86	75	47	43	58
errors	0	0	52	29	1

Table 2: ft Prolog, ft C, LJT, STRIP and PITP on ILTP v1.1.1 formulas

	SYJ202+1 provable	SYJ205+1 provable	SYJ206+1 provable	SYJ207+1 refutable	SYJ208+1 refutable	SYJ209+1 refutable	SYJ211+1 refutable	SYJ212+1 refutable
ft Prolog	07 (516.55)	08 (60.26)	10 (144.5)	07 (358.05)	08 (65.41)	10 (543.09)	04 (66.62)	20 (0.01)
ft C	07 (76.3)	09 (85.84)	11 (481.98)	07 (51.13)	17 (81.41)	10 (96.99)	04 (17.25)	20 (0.01)
LJT	02 (0.09)	20 (0.01)	05 (0.01)	03 (2.64)	08 (0.18)	10 (461.27)	08 (546.46)	07 (204.98)
STRIP	06 (11.28)	14 (267.39)	20 (37.64)	04 (9.3)	06 (0.24)	10 (132.55)	09 (97.63)	20 (36.79)
PITP	09 (595.79)	20 (0.01)	20 (4.07)	04 (11.11)	08 (83.66)	10 (280.47)	20 (526.16)	11 (528.08)

Table 3: ILTP v1.1.1 formulas solved by classes

missing, results in Table 2 (this table is taken from `http://www.iltp.de/download/-ILTP-v1.1.1-prop-comparison.txt`) shows that PITP outperforms the known theorem provers on ILTP v1.1.1 library. Within 10 minutes PITP decides 215 out of 274 formulas of ILTP v1.1.1 The library divides the formulas in several families. Every family contains formulas sharing the same pattern of increasing complexity. In Table 3

	SYJ201+1	SYJ202+1	SYJ207+1	SYJ208+1	SYJ209+1	SYJ211+1	SYJ212+1
PITP none	20 (1.29)	03 (0.01)	04 (43.77)	04 (2.50)	10 (596.55)	20 (526.94)	11 (527.72)
PITP -opt1	20 (0.03)	08 (44.59)	04 (44.76)	08 (93.60)	10 (325.93)	20 (558.11)	11 (548.01)
PITP -opt2	20 (1.67)	03 (0.01)	04 (12.18)	04 (2.37)	10 (311.37)	19 (293.34)	10 (88.92)
PITP -opt3	20 (0.03)	08 (44.21)	04 (11.36)	08 (94.30)	10 (591.68)	19 (291.18)	10 (92.05)
PITP ALL	20 (0.03)	08 (45.30)	04 (12.74)	08 (90.11)	10 (297.83)	19 (313.11)	10 (93.18)

Table 4: Comparison between PITP optimizations

(this table is taken from `http://www.iltp.de/`) for every family (some families of ILTP v1.1.1 are missing because they are decided within 1s by all provers) we report the index of the largest formula which every prover is able to decide within 600s CPU time and in parenthesis the CPU time necessary to solve such a formula. PITP solves all the formulas in three families and it is the best prover in three families (SYJ202+1, SYJ206+1, SYJ211+1), ft-C solves all the formulas of SYJ212+1 and it is the best prover in four families (SYJ207+1, SYJ208+1, SYJ210+1, SYJ212+1), finally STRIP solves all the formulas in two families but in no class is it the best prover. Finally we run PITP on our Xeon 3.2GHz machine to evaluate the effect of using the optimizations described above. It is well known to people working in ATP that an optimization can be effective for one class of formulas and be negative for other classes. In Table 4 we compare different optimizations and give the results of their use on some classes of ILTP v1.1.1 formulas. First, PITP without optimizations outperforms the other versions of PITP on the families SYJ211+1 and SYJ212+1. To give an idea of the overhead of the optimizations on formulas where such optimizations do not apply, PITP without optimizations solves the 19th formula of SYJ211+1 in 247.19 seconds and the 10th formula of SYJ212+1 in 76.55 seconds. Among the optimizations the most important seems **opt2**. When **opt2** is not active the performances decrease. Thus even if this optimization can be used only on particular classes of formulas, it dramatically influences the performances (in our opinion this gives an idea of the great importance of bounding branching in propositional intuitionistic logic). With regard to the other optimizations, there are some advantages in some classes and disadvantages in others. In table 5 we provide the results of the comparison between PITP and STRIP on twelve thousand random formulas with three hundred connectives (since the performance of ft-C was worse than STRIP and PITP, table 5 lacks of ft-C). Given the time limit of five minutes, STRIP does not decide 780 formulas, PITP does not decide 16 formulas.

References

[AFF$^+$06] A. Avellone, M. Ferrari, C. Fiorentini, G. Fiorino, and U. Moscato. Esbc: an application for computing stabilization bounds. *ENTCS*, 153(1):23–33, 2006.

[AFM04] A. Avellone, G. Fiorino, and U. Moscato. A new O($n \lg n$)-space decision procedure for propositional intuitionistic logic. In Andrei Voronkov Matthias Baaz, Johann Makowsky, editor, *LPAR 2002: Short Contributions, CSL 2003: Extended Posters*, volume VIII of *Kurt Gödel Society, Collegium Logicum*, 2004.

[Bak95] A. B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD thesis, University of Oregon, 1995.

[BC04] Yves Bertot and P. (Pierre) Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Texts in theoretical computer science. Springer-Verlag, pub-SV:adr, 2004.

[Con86] R. Constable. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice–Hall, Englewood Cliffs, New Jersey, 1986.

[CZ97] A. Chagrov and M. Zakharyaschev. *Modal Logic.* Oxford University Press, 1997.

[Dyc92] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3):795–807, 1992.

[ES99] Uwe Egly and Stephan Schmitt. On intuitionistic proof transformations, their complexity, and application to constructive program synthesis. *Fundam. Inform.*, 39(1-2):59–83, 1999.

[FFO02] M. Ferrari, C. Fiorentini, and M. Ornaghi. Extracting exact time bounds from logical proofs. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001, Selected Papers*, volume 2372 of *Lecture Notes in Computer Science*, pages 245–265. Springer-Verlag, 2002.

[Fit69] M.C. Fitting. *Intuitionistic Logic, Model Theory and Forcing.* North-Holland, 1969.

[Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, 1995.

[Hud93] J. Hudelmaier. An $O(n \log n)$-SPACE decision procedure for intuitionistic propositional logic. *Journal of Logic and Computation*, 3(1):63–75, 1993.

[Men00] M. Mendler. Timing analysis of combinational circuits in intuitionistic propositional logic. *Formal Methods in System Design*, 17(1):5–37, 2000.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory. Bibliopolis, Napoli, 1984.

[MMO97] P. Miglioli, U. Moscato, and M. Ornaghi. Avoiding duplications in tableau systems for intuitionistic logic and Kuroda logic. *Logic Journal of the IGPL*, 5(1):145–167, 1997.

[ROK06] Thomas Raths, Jens Otten, and Christoph Kreitz. The ILTP problem library for intuitionistic logic. release v1.1. *To appear in Journal of Automated Reasoning*, 2006.

	provable			
	0-1s	1-10s	10-200s	200-300s
STRIP	200	19	15	0
PITP	192	36	11	2

	unprovable			
	0-1s	1-10s	10-200s	200-300s
STRIP	10139	404	394	49
PITP	11663	49	27	4

	$> 300s$
STRIP	780
PITP	16

Table 5: PITP and STRIP on random generated formulas

[Sta79] R. Statman. Intuitionistic logic is polynomial-space complete. *Theoretical Computer Science*, 9(1):67–72, 1979.

[Wei98] Klaus Weich. Decision procedures for intuitionistic propositional logic by program extraction. In *TABLEAUX*, pages 292–306, 1998.

LIFT-UP: Lifted First-Order Planning Under Uncertainty

Steffen Hölldobler and Olga Skvortsova
International Center for Computational Logic
Technische Universität Dresden, Dresden, Germany
{sh,skvortsova}@iccl.tu-dresden.de

Abstract

We present a new approach for solving first-order Markov decision processes combining first-order state abstraction and heuristic search. In contrast to existing systems, which start with propositionalizing the decision process and then perform state abstraction on its propositionalized version we apply state abstraction directly on the decision process avoiding propositionalization. Secondly, guided by an admissible heuristic, the search is restricted to those states that are reachable from the initial state. We demonstrate the usefulness of the above techniques for solving first-order Markov decision processes within a domain dependent system called FluCaP which participated in the probabilistic track of the 2004 International Planning Competition. Working toward a domain independent implementation we present novel approaches to θ-subsumption involving literal and object contexts.

1 Introduction

We are interested in solving probabilistic planning problems, i. e. planning problems, where the execution of an action leads to the desired effects only with a certain probability. For such problems, Markov decision processes have been adopted as a representational and computational model in much recent work, e.g., by [BBS95]. They are usually solved using the so-called dynamic programming principle [BDH99] employing a value iteration algorithm. Classical dynamic programming algorithms explicitly enumerate the state space and are thus exponenetial. In recent years several methods have been developed which avoid an explicit enuration of the state space. The most prominent are state abstraction [BDH99], heuristic search (e. g. [BBS95, DKKN95]) and a combination of both as used, for example, in symbolic LAO* [FH02].

A common feature of these approaches is that a Markov decision process is propositionalized before state abstraction techniques and heuristic search algorithms are applied within a value iteration algorithm. Unfortunately, the propositionalization step itself may increase the problem significantly. To overcome this problem, it was first proposed in [BRP01] to solve first-order Markov decision processes by applying a first-order value iteration algorithm and first-order state abstraction techniques. Whereas this symbolic dynamic programming approach was rooted in a version of the Situation Calculus [Rei91], we have reformulated and extended these ideas in a variant of the fluent calculus [HS04]. In this system, which is now called LIFT-UP, lifted first-order planning under uncertainty can be performed.

In the LIFT-UP system, states and actions are expressed in the language of the fluent calculus [HS90], which is slightly extended to handle probabilities. In addition, value functions and policies are represented by constructing first-order formulas which partition the state space into clusters, referred to as *abstract states*. Then, value iteration can be performed on top of these clusters, obviating the need for explicit state enumeration. This allows the solution of first-order Markov decision processes without requiring explicit state enumeration or propositionalization. In addition, heuristics are used to guide the search and normalization techniques are applied to eliminate redundant states. The LIFT-UP approach can thus be viewed as a first-order generalization of symbolic LAO* or, alternatively, as symbolic dynamic programming enhanced by heuristic search and state space normalization.

To evaluate the LIFT-UP system we have developed a domain-dependent implementation called FluCaP. It can solve probabilistic blocksword problems as they appeared, for example, in the colored blocksworld domain of the 2004 International Planning Competition. FluCaP is quite successful and outperforming other systems on truly first-order problems. On the other hand and working towards a domain-independent implementation we have studied θ-subsumption algorithms. θ-subsumption problems arise at various places in the LIFT-UP system: The normalization process requires to check whether one abstract state subsumes another one; the check whether an action is applicable to some abstract state and the computation of set of the successor or predecessor states also requires subsumption. One should observe that the latter application requires to compute a complete set of substitutions.

In this paper we give an overview of the LIFT-UP approach.

2 First-order Markov Decision Processes

A *Markov decision process*, is a tuple $(\mathcal{Z}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{C})$, where $\mathcal{Z}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, and $\mathcal{P} : \mathcal{Z} \times \mathcal{Z} \times \mathcal{A} \to [0,1]$, written $\mathcal{P}(z'|z,a)$, specifies transition probabilities. In particular, $\mathcal{P}(z'|z,a)$ denotes the probability of ending up at state z' given that the agent was in state z and action a was executed. $\mathcal{R} : \mathcal{Z} \to \mathbb{R}$ is a real-valued reward function associating with each state z its immediate utility $\mathcal{R}(z)$. $\mathcal{C} : \mathcal{A} \to \mathbb{R}$ is a real-valued cost function associating a cost $\mathcal{C}(a)$ with each action a. A *sequential decision problem* consists of a Markov decision process and is the problem of finding a policy $\pi : \mathcal{Z} \to \mathcal{A}$ that maximizes the total expected discounted reward received when executing the policy π over an infinite (or indefinite) horizon. A Markov decision process is said to be *first-order* if the expressions used to define $\mathcal{Z}$, $\mathcal{A}$ and $\mathcal{P}$ are first-order.

The *value* $V_\pi(z)$ of a state z with respect to the policy π is defined as

$$V_\pi(z) = \mathcal{R}(z) + \mathcal{C}(\pi(z)) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z'|z, \pi(z)) V_\pi(z'),$$

where $0 \leq \gamma \leq 1$ is a discount factor. We take γ equal to 1 for indefinite-horizon problems only, i. e. when a goal is reached the system enters an absorbing state in which no further rewards or costs are accrued. A value function V is set to be *optimal* if it satisfies

$$\mathcal{R}(z) + \max_{a \in \mathcal{A}} \{\mathcal{C}(a) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z'|z,a) V^*(z')\} \ ,$$

for each $z \in \mathcal{Z}$; in this case the value function is usually denoted by $V^*(z)$. The optimal policy is extracted from the optimal value function.

We assume that planning problems meet the following requirements:

1. Each problem has a goal statement, identifying a set of absorbing goal states.
2. A positive reward is associated with each action ending in a goal state; otherwise it is 0.
3. A cost is associated with each action.
4. A "done" action is available in all states.

The "done" action can be used to end any further accumulation of reward. Together, these conditions ensure that an MDP model of a planning problem is a positive bounded model as described by [Put94]. Such planning problems are also often called stochastic shortest path problems.

3 Probabilistic Fluent Calculus

States, actions, transition probabilities, cost and reward function are specified in a probabilistic and sorted extension of the fluent calculus [HS90, Thi98].

Fluents and States Let Σ denote a set of function symbols containing the binary function symbol $\circ$ and the nullary function symbol 1. $\circ$ is an AC1-symbol with 1 as unit element. Let $\Sigma^- = \Sigma \setminus \{\circ, 1\}$. Non-variable Σ^--terms are called *fluents*. Let $f(t_1, \ldots, t_n)$ be a fluent. The terms t_i, $1 \le i \le n$ are called *objects*. A *state* is a finite set of ground fluents. Let $\mathcal{D}$ be the set of all states.

Fluent Terms and Abstract States *Fluent terms* are defined inductively as follows: 1 is a fluent term; each fluent is a fluent term; if G_1 and G_2 are fluent terms, then so is $G_1 \circ G_2$. Let $\mathcal{F}$ be the set of all fluent terms. We assume that each fluent term obeys the *singularity condition*: each fluent may occur at most once in a fluent term. Because of the latter, there is a bijection $\cdot^M$ between ground fluent terms and states. Some care must be taken when instantiating a non-ground fluent term F by a substitution θ because $F\theta$ may violate the singularity condition. A substitution θ is *allowed* for fluent term F if $F\theta$ meets the singularity condition.

Abstract states are expressions of the form F or $F \circ X$, where F is a fluent term and X is a variable of sort fluent term. Let $\mathcal{S}$ denote the set of abstract states. Abstract states denote sets of states as defined by the mapping $\cdot^I : \mathcal{S} \to 2^{\mathcal{D}}$: Let Z be an abstract state. Then

$$[Z]^I = \{[Z\theta]^M \mid \theta \text{ is an allowed grounding substitution for } Z\}.$$

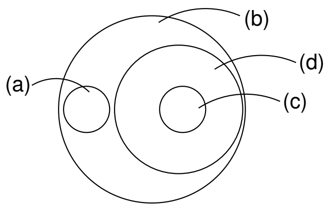


Figure 1: The interpretations of the abstract states (a) $Z_1 = on(X_1, a) \circ on(a, table)$, (b) $Z_2 = on(X_2, a) \circ on(a, table) \circ Y_2$, (c) $Z_3 = on(X_3, a) \circ on(a, table) \circ clear(X_3)$ and (d) $Z_4 = on(X_4, a) \circ on(a, table) \circ clear(X_4) \circ Y_4$, where a is an object denoting a block, *table* is an object denoting a table, X_1, X_2, X_3 and X_4 are variables of sort object, Y_2 and Y_4 are variables of sort fluent term, $on(X_i, a)$, $i = 1 \ldots 4$, is a fluent denoting that some block X_i is on a and $clear(X_i)$, $i = 3, 4$, is a fluent denoting that block X_i is clear.

This is illustrated in Figure 1. In other words, abstract states are characterized by means of positive conditions that must hold in each ground instance thereof and, thus, they represent clusters of states. In this way, abstract states embody a form of state space abstraction, which is called *first-order state abstraction*.

As a running example, we consider problems taken from the colored Blocksworld scenario, which is an extension of the classical Blocksworld scenario in the sense that along with the unique identifier, each block is now assigned a specific color. Thus, a state description provides an arrangement of colors instead of an arrangement of blocks. For example, a state Z defined as a fluent term:

$$Z = red(X_0) \circ green(X_1) \circ blue(X_2) \circ red(X_3) \circ red(X_4) \circ \\ red(X_5) \circ green(X_6) \circ green(X_7) \circ Tower(X_0, \ldots, X_7) \ ,$$

specifies a tower that is comprised of eigth colored blocks.

Subsumption Let Z_1 and Z_2 be abstract states. Then Z_1 is *subsumed* by Z_2, in symbols $Z_1 \sqsubseteq Z_2$, if there exists an allowed substitution θ such that $Z_2\theta =_{AC1} Z_1$. Intuitively, Z_1 is subsumed by Z_2 iff $Z_1^I \subseteq Z_2^I$. In the LIFT-UP system we are often concerned with the problem of finding a complete set of allowed substitutions solving the AC1-matching problem $Z_2\theta =_{AC1} Z_1$.

For example, consider the abstract states mentioned in Figure 1. Then, $Z_1 \sqsubseteq Z_2$ with $\theta = \{X_2 \mapsto X_1, Y_2 \mapsto 1\}$, $Z_3 \sqsubseteq Z_2$ with $\theta = \{X_2 \mapsto X_3, Y_2 \mapsto clear(X_3)\}$. However, $Z_1 \not\sqsubseteq Z_3$ and $Z_3 \not\sqsubseteq Z_1$.

Actions Let Σ_a denote a set of action names, where $\Sigma_a \cap \Sigma = \emptyset$. An *action space* $\mathcal{A}$ is a set of expressions of the form $(a(X_1, \ldots, X_n), C, E)$, where $a \in \Sigma_a$, X_i, $1 \leq i \leq n$, are variables or constants, $C \in \mathcal{F}$ called *precondition* and $E \in \mathcal{F}$ called *effect* of the *action* $a(X_1, \ldots, X_n)$. E.g., a pickup-action in the blockworld can be specified by

$$(pickup\,(X, Y),\ on(X, Y) \circ clear(X) \circ empty,\ holding(X) \circ clear(Y)),$$

where *empty* denotes that the robot arm is empty and $holding(X)$ that the block X is in the gripper. For simplicity, we will often supress parameters, preconditions and effects of an action $(a(X_1, \ldots, X_n), C, E)$ and refer to it as a instead.

Nature's Choice and Probabilities In analogy to the approach in [BRP01] stochastic actions are decomposed into deterministic primitives under nature's control, referred to as *nature's choices*. It can be modelled with the help of a binary relation symbol *choice* as follows: Consider the action $\mathit{pickup}(X,Y)$:

$$\mathit{choice}(\mathit{pickup}(X,Y),a) \leftrightarrow \ (a = \mathit{pickupS}(X,Y) \vee a = \mathit{pickupF}(X,Y)),$$

where *pickupS* and *pickupF* define two nature's choices for action *pickup*, viz., that it succeeds or fails. For simplicity, we denote the set of nature's choices of an action a as $\mathit{Ch}(a) := \{a_j | \mathit{choice}(a, a_j)\}$.

For each of nature's choices a_j associated with an action a we define the probability $\mathit{prob}(a_j, a, Z)$ denoting the probability with which one of nature's choices a_j is chosen in a state Z. For example,

$$\mathit{prob}(\mathit{pickupS}(X,Y), \mathit{pickup}(X,Y), Z) = .75$$

states that the probability for the successful execution of the *pickup* action in state Z is .75. We require that for each action the probabilities of all its nature's choices sum up to 1.

Rewards and Costs Reward and cost functions are defined for abstract states using the unary relation symbols *reward* and *cost*. For example, we might want to give a reward of 500 to all states in which some block X is on block a and 0, otherwise:

$$\begin{array}{lcl} \mathit{reward}(Z) = 500 & \leftrightarrow & Z \sqsubseteq (on(X,a), \emptyset), \\ \mathit{reward}(Z) = 0 & \leftrightarrow & Z \not\sqsubseteq (on(X,a), \emptyset). \end{array}$$

In other words, the state space is divided into two abstract states depending on whether or not, a block X is on block a. Likewise, value functions can be specified with respect to the abstract states only. Action costs can be analogously defined. E. g., with

$$\mathit{cost}(\mathit{pickup}(X,Y)) = 3$$

the execution of the *pickup*-action is penalized with 3.

Forward and Backward Application of Actions An action $(a(X_1,\ldots,X_n), C, E)$ is *forward applicable with* θ to an abstract state $Z \in \mathcal{S}$, denoted as $\mathit{forward}(Z,a,\theta)$, if $(C \circ U)\theta =_{AC1} Z$, where U is a new variable of sort fluent term and θ is an allowed substitution. If applicable, then the action *progresses to* or *yields* the state $(E \circ U)\theta$. In this case, $(E \circ U)\theta$ is called *successor state* of Z and denoted as $\mathit{succ}(Z,a,\theta)$.

An action $(a(X_1,\ldots,X_n), C, E)$ is *backward applicable with* θ to an abstract state $Z \in \mathcal{S}$, denoted as $\mathit{backward}(Z,a,\theta)$, if $(E \circ U)\theta =_{AC1} Z$, where U is a new variable of sort fluent term and θ is an allowed substitution. If applicable, then the action *regresses to* the state $(C \circ U)\theta$. In this case, $(C \circ U)\theta$ is called *predecessor state* of Z and denoted as $\mathit{pred}(Z,a,\theta)$.

One should observe that the AC1-matching problems involved in the application of actions are subsumption problems, viz. $Z \sqsubseteq (C \circ U)$ and $Z \sqsubseteq (E \circ U)$. Moreover, in order to determine all possible successor or predecessor states of some state with respect to some action we have to compute complete sets of allowed substitutions solving the corresponding subsumption problems.

```
$policyExpansion(\pi, \mathcal{S}^0, G)$
 $E := F := \emptyset$
 $from := \mathcal{S}^0$
 repeat
  $to := \bigcup_{Z \in from} \bigcup_{a_j \in Ch(a)} \{succ(Z, a_j, \theta)\}$,
  where $(a, \theta) := \pi(Z)$
  $F := F \cup (to - G)$
  $E := E \cup from$
  $from := to \cap G - E$
 until $(from = \emptyset)$
 $E := E \cup F$
 $G := G \cup F$
 return $(E, F, G)$

FOVI$(E, \mathcal{A}, prob, reward, cost, \gamma, V)$
 repeat
  $V' := V$
  loop for each $Z \in E$
   loop for each $a \in \mathcal{A}$
    loop for each $\theta$ such that $forward(Z, a, \theta)$
     $Q(Z, a, \theta) := reward(Z) + cost(a) +$
        $\gamma \sum_{a_j \in Ch(a)} prob(a_j, a, Z) \cdot V'(succ(Z, a_j, \theta))$
    end loop
   end loop
   $V(Z) := \max_{(a,\theta)} Q(Z, a, \theta)$
  end loop
  $V := normalize(V)$
  $r := \|V - V'\|$
 until stopping criterion
 $\pi := extractPolicy(V)$
 return $(V, \pi, r)$

FOLAO$^*(\mathcal{A}, prob, reward, cost, \gamma, \mathcal{S}^0, h, \varepsilon)$
 $V := h$
 $G := \emptyset$
 For each $Z \in \mathcal{S}^0$, initialize $\pi$ with an arbitrary action
 repeat
   $(E, F, G) := policyExpansion(\pi, \mathcal{S}^0, G)$
   $(V, \pi, r) :=$ FOVI$(E, \mathcal{A}, prob, reward, cost, \gamma, V)$
 until $(F = \emptyset)$ and $r \leq \varepsilon$
 return $(\pi, V)$
```

Figure 2: LIFT-UP algorithm.

4 LIFT-UP Algorithm

In order to solve first-order MDPs, we have developed a new algorithm that combines heuristic search and first-order state abstraction techniques.

Our algorithm, referred to as LIFT-UP, can be seen as a generalization of the symbolic LAO* algorithm by [FH02]. Given an initial state, LIFT-UP uses an admissible heuristic to focus computation on the parts of the state space that are reachable from the initial state. Moreover, it specifies MDP components, value functions, policies, and admissible heuristics using a first-order language of the Probabilistic Fluent Calculus. This allows LIFT-UP to manipulate abstract states instead of individual states. The algorithm itself is presented in Figure 2.

As symbolic LAO*, LIFT-UP has two phases that alternate until a complete solution is found, which is guaranteed to be optimal. First, it expands the best partial policy and evaluates the states on its fringe using an admissible heuristic function. Then it performs dynamic programming on the states visited by the best partial policy, to update their values and possibly revise the current best partial policy. We note that we focus on partial policies that map a subcollection of states into actions.

In the policy expansion step, we perform reachability analysis to find the set F of states that have not yet been expanded, but are reachable from the set $\mathcal{S}^0$ of initial states by following the partial policy π. The set of states G contains states that have been expanded so far. By expanding a partial policy we mean that it will be defined for a larger set of states in the dynamic programming step.

In symbolic LAO*, reachability analysis is performed on propositional algebraic decision diagrams (ADDs). Therefore, an additional preprocessing of a first-order MDP is required at the outset of any solution attempt. This preprocessing involves propositionalization of the first-order structure of an MDP, viz., instantiation of the MDP components with all possible combinations of domain objects. Whereas, LIFT-UP relies on the lifted first-order reasoning, that is, computations are kept on the first-order level avoiding propositionalization. In particular, action applicability check and computation of successors as well as predecessors are accomplished on abstract states directly.

In the dynamic programming step of LIFT-UP, we employ a modified first-order value iteration algorithm (FOVI) that computes the value only on those states which are reachable from the initial states. More precisely, we call FOVI on the set E of states that are visited by the best current partial policy. In this way, we improve the efficiency of the original FOVI algorithm by [HS04] by using symbolic dynamic programming together with reachability analysis.

Given a FOMDP and a value function represented in PFC, FOVI returns the best partial value function V, the best partial policy π and the residual r. In order to update the values of the states Z in E, we assign the values from the current value function to the successors of Z. We compute successors with respect to all nature's choices a_j. The residual r is computed as the absolute value of the largest difference between the current and the newly computed value functions V' and V, respectively. We note that the newly computed value function V is taken in its normalized form, i.e., as a result of the *normalize* procedure that will be described in Section 4.2.1. Extraction of a best partial policy π is straightforward: One simply needs to extract the maximizing actions from the best partial value function V.

As with symbolic LAO*, LIFT-UP converges to an ε-optimal policy when three conditions are met: (1) its current policy does not have any unexpanded states, (2) the residual r is less than the predefined threshold ε, and (3) the value function is initialized with an admissible heuristic. The original convergence proofs for LAO* and symbolic LAO* by [HZ01] carry over in a straightforward way to LIFT-UP.

When calling LIFT-UP, we initialize the value function with an admissible heuristic function h that focuses the search on a subset of reachable states. A simple way to create an admissible heuristic is to use dynamic programming to compute an approximate value function. Therefore, in order to obtain an admissible heuristic h in LIFT-UP, we perform several iterations of the original FOVI. We start the algorithm on an initial value function that is admissible. Since each step of FOVI preserves admissibility, the resulting value function is admissible as well. The initial value function assigns the goal reward to each state thereby overestimating the optimal value, since the goal reward is the maximal possible reward.

Since all computations in LIFT-UP are performed on abstract states instead of individual states, FOMDPs are solved avoiding explicit state and action enumeration and propositionalization. Lifted first-order reasoning leads to better performance of LIFT-UP in comparison to symbolic LAO*, as shown in Section 5.2.

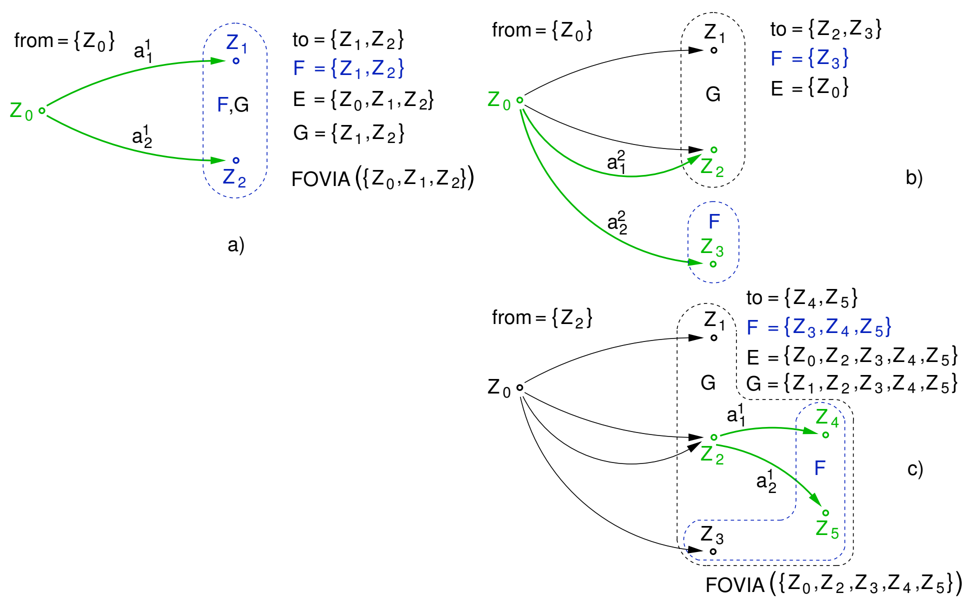


Figure 3: Policy Expansion.

4.1 Policy Expansion

We illustrate the policy expansion procedure in LIFT-UP by means of an example. Assume that we start from the initial state Z_0 and two nondeterministic actions a^1 and a^2 are applicable in Z_0, each having two outcomes a^1_1, a^1_2 and a^2_1, a^2_2, respectively. Without loss of generality, we assume that the current best policy π chooses a^1 as an optimal action at state Z_0. We construct the successors Z_1 and Z_2 of Z_0 with respect to both outcomes a^1_1 and a^1_2 of the action a^1. The fringe set F as well as the set G of states expanded so far contain the states Z_1 and Z_2 only, whereas, the set E of states visited by the best current partial policy gets the state Z_0 in addition. See Figure 3a. In the next step, FOVI is performed on the set E. We assume that the values have been updated in such a way that a^2 becomes an optimal action in Z_0. Thus, the successors of Z_0 have to be recomputed with respect to the optimal action a^2. See Figure 3b.

One should observe that one of the a^2-successors of Z_0, namely Z_2, is an element of the set G and thus, it has been contained already in the fringe F during the previous expansion step. Hence, the state Z_2 should be expanded and its value recomputed. This is shown in Figure 3c, where states Z_4 and Z_5 are a^1-successors of Z_2, under assumption that a^1 is an optimal action in Z_2. As a result, the fringe set F contains the newly discovered states Z_3, Z_4 and Z_5 and we perform FOVI on $E = \{Z_0, Z_2, Z_3, Z_4, Z_5\}$. The state Z_1 is not contained in E, because it does not belong to the best current partial policy, and the dynamic programming step is performed only on the states that were visited by the best current partial policy.

N	Number of states		Time, msec		Runtime, msec	Runtime w/o norm, msec
	$\mathcal{S}_{\mathsf{update}}$	$\mathcal{S}_{\mathsf{norm}}$	Update	Norm		
0	9	6	144	1	145	144
1	24	14	393	3	396	593
2	94	23	884	12	896	2219
3	129	33	1377	16	1393	13293
4	328	39	2079	46	2125	77514
5	361	48	2519	51	2570	805753
6	604	52	3268	107	3375	n/a
7	627	54	3534	110	3644	n/a
8	795	56	3873	157	4030	n/a
9	811	59	4131	154	4285	n/a

Table 1: Representative timing results for the first ten iterations of the first-order value iteration with the normalization procedure switched on or off.

4.2 First-order Value Iteration

The first-order value iteration algorithm (FOVI) produces a first-order representation of the optimal value function and policy by exploiting the logical structure of a first-order MDP. Thus, FOVI can be seen as a first-order counterpart of the classical value iteration algorithm by [Bel57].

In LIFT-UP, the first-order value iteration algorithm serves two purposes: First, we perform several iterations of FOVI in order to create an admissible heuristic h in LIFT-UP. Second, in the dynamic programming step of LIFT-UP, we apply FOVI on the states visited by the best partial policy in order to update their values and possibly revise the current best partial policy.

4.2.1 Normalization

It was already mentioned by several authors that value iteration adds a dramatic computational overhead to a solution technique for first-order MDPs if no care about redundant computations is taken [BRP01, HS04].

Recently, there have been proposed an automated normalization procedure that, given a state space, delivers an equivalent one that contains no redundancy [HS04]. This procedure, referred to as *normalize* in the LIFT-UP algorithm, is always called before the value function is transmitted to the next iteration step, thereby preventing the propagation of redundancy to the next computation steps. The technique employs the notion of the subsumption relation defined in Section 3. Informally, given two abstract states Z_1 and Z_2 such that $Z_1 \sqsubseteq Z_2$ and the values associated to states are identical, Z_1 can be easily removed from the state space because it contains redundant information.

Table 1 illustrates the importance of the normalization algorithm by providing some representative timing results for the first ten iterations of the first-order value iteration. The experiments were carried out on the problem taken from the colored Blocksworld scenario consisting of ten blocks. Even on such a relatively simple problem FOVI with the normalization switched off does not scale beyond the sixth iteration.

The results in Table 1 demonstrate that the normalization during some iteration of FOVI dramatically shrinks the computational effort during the next iterations. The columns labelled $\mathcal{S}_{\mathsf{update}}$ and $\mathcal{S}_{\mathsf{norm}}$ show the size of the state space after performing the value updates and the normalization, respectively. For example, the normalization factor, i.e., the ratio between the number $\mathcal{S}_{\mathsf{update}}$ of states obtained after performing one update step and the number $\mathcal{S}_{\mathsf{norm}}$ of states obtained after performing the normalization step, at the seventh iteration is 11.6. This means that more than ninety percent of the state space contained redundant information. The fourth and fifth columns in Table 1 contain the time Update and Norm spent on performing value updates and on the normalization, respectively. The total runtime Runtime, when the normalization is switched on, is given in the sixth column. The seventh column labelled Runtime w/o norm depicts the total runtime of FOVI when the normalization is switched off. If we would sum up all values in the seventh column and the values in the sixth column up to the sixth iteration inclusively, subtract the latter from the former and divide the result by the total time Norm needed for performing normalization during the first six iterations, then we would obtain the normalization gain of about three orders of magnitude.

5 The Planning System FluCaP

To evaluate the LIFT-UP approach we have developed a domain-dependent implementation called FluCaP. It can solve probabilistic Blocksworld problems as they appeared, for example, in the colored Blocksworld domain of the 2004 International Planning Competition.

5.1 Domain-dependent Optimizations

So far, we have presented a general theory of LIFT-UP for finding solutions in uncertain planning environments which are represented as first-order MDPs. However, several domain-driven optimizations or relaxations have been posed on the general theory. As a result, FluCaP has demonstrated a competitive computational behaviour.

Action Applicability Since in the Blocksworld domain, all states were fully specified, abstract states were described as fluent terms only. This allows to relax the forward and backward action applicability conditions. Since the cases are symmetric, we will concentrate on the forward action applicability condition that was initially defined as: An action $(a(X_1, \ldots, X_n), C, E)$ is *forward applicable with* θ to an abstract state $Z \in \mathcal{S}$, denoted as $\mathit{forward}(Z, a, \theta)$, if $(C \circ U)\theta =_{AC1} Z$, where U is a new variable of sort fluent term and θ is an allowed substitution. Since C and Z are fluent terms under the singularity condition, the aforementioned AC1-matching problem can be transformed into the θ-subsumption problem [Rob65].

Moreover, we optimize the obtained θ-subsumption problem further. Since a state description in a colored Blocksworld represents a number of towers, we compare towers of blocks and their color distributions instead of matching respective fluent terms. The experiments have shown that it is much faster to manipulate with towers rather than with fluent terms. For example, assume that an action precondition contains a fluent $\mathit{clear}(X)$. Let a state describe three towers of blocks. By inspecting the uppermost blocks in the towers, we conclude that there are only three blocks, which satisfy the precondition. It would be interesting to check whether this optimization technique can be successfully applied in other planning domains as well.

Meanwhile, in Section 6, we present some results towards efficient domain-independent solution methods for the θ-subsumption problem.

Normalization The similar situation occurs in the case of normalization which relies on the subsumption relation defined in Section 3. The AC1-matching problem underlying the subsumption relation reduces to the θ-subsumption problem.

Again, it is much faster to operate on towers rather than on fluent terms. For example, assume that one state describes two towers of four and three blocks, respectively. Another state also describes two towers but of five and two blocks, respectively. In order to decide, whether one state subsumes another one we try to match the corresponding towers and their color distributions. As experiments have shown, this optimization speeds up the normalization immensely.

5.2 Experimental Evaluation

We demonstrate the advantages of combining the heuristic search together with first-order state abstraction on a FLUCAP system, that has successfully entered the domain-dependent track of the probabilistic part of the 2004 International Planning Competition (IPC'2004). The experimental results were all obtained using RedHat Linux running on a 3.4GHz Pentium IV machine with 3GB of RAM.

In Table 2, we present the performance comparison of FLUCAP together with symbolic LAO* on examples taken from the colored Blocksworld (BW) scenario. Our main objective was to investigate whether first-order state abstraction using logic could improve the computational behaviour of a planning system for solving FOMDPs. The colored BW problems were our main interest since they were the only ones represented in first-order terms and hence the only ones that allowed us to make use of the first-order state abstraction.

At the outset of solving a colored BW problem, symbolic LAO* starts by propositionalizing its components, namely, the goal statement and actions. Only after that, the abstraction using propositional ADDs is applied. In contrast, FLUCAP performs first-order abstraction on a colored BW problem directly, avoiding unnecessary grounding. In the following, we show how an abstraction technique affects the computation of a heuristic function. To create an admissible heuristic, FLUCAP performs twenty iterations of FOVI and symbolic LAO* performs twenty iterations of an approximate value iteration algorithm similar to APRICODD by [SAHB00]. The columns labelled H.time and NAS show the time needed for computing a heuristic function and the number of abstract states it covers, respectively. In comparison to FLUCAP, symbolic LAO* needs to evaluate fewer abstract states in the heuristic function but takes considerably more time. One can conclude that abstract states in symbolic LAO* enjoy more complex structure than those in FLUCAP.

We note that, in comparison to FOVI, FLUCAP restricts the value iteration to a smaller state space. Intuitively, the value function, which is delivered by FOVI, covers a larger state space, because the time that is allocated for the heuristic search in FLUCAP is now used for performing additional iterations in FOVI. The results in the column

Problem		Total av. reward				Total time, sec.				H.time, sec.		NAS			NGS, $\times 10^3$		%
B	C	LAO*	FluCaP	FOVI	FluCaP$^-$	LAO*	FluCaP	FOVI	FluCaP$^-$	LAO*	FluCaP	LAO*	FluCaP	FOVI	LAO*	FluCaP	FluCaP
	4	494	494	494	494	22.3	22.0	23.4	31.1	8.7	4.2	35	410	1077	0.86	0.82	2.7
5	3	496	495	495	496	23.1	17.8	22.7	25.1	9.5	1.3	34	172	687	0.86	0.68	2.1
	2	496	495	495	495	27.3	11.7	15.7	16.5	12.7	0.3	32	55	278	0.86	0.66	1.9
	4	493	493	493	493	137.6	78.5	261.6	285.4	76.7	21.0	68	1061	3847	7.05	4.24	3.1
6	3	493	492	493	492	150.5	33.0	119.1	128.5	85.0	9.3	82	539	1738	7.05	6.50	2.3
	2	495	494	495	496	221.3	16.6	56.4	63.3	135.0	1.2	46	130	902	7.05	6.24	2.0
	4	492	491	491	491	1644	198.1	2776	n/a	757.0	171.3	143	2953	12014	65.9	23.6	3.5
7	3	494	494	494	494	1265	161.6	1809	2813	718.3	143.6	112	2133	7591	65.9	51.2	2.4
	2	494	494	494	494	2210	27.3	317.7	443.6	1241	12.3	101	425	2109	65.9	61.2	2.0
	4	n/a	490	n/a	n/a	n/a	1212	n/a	n/a	n/a	804.1	n/a	8328	n/a	n/a	66.6	4.1
8	3	n/a	490	n/a	n/a	n/a	598.5	n/a	n/a	n/a	301.2	n/a	3956	n/a	n/a	379.7	3.0
	2	n/a	492	n/a	n/a	n/a	215.3	1908	n/a	n/a	153.2	n/a	2019	7251	n/a	1121	2.3
15	3	n/a	486	n/a	n/a	n/a	1809	n/a	n/a	n/a	1733	n/a	7276	n/a	n/a	$1.2 \cdot 10^7$	5.7
17	4	n/a	481	n/a	n/a	n/a	3548	n/a	n/a	n/a	1751	n/a	15225	n/a	n/a	$2.5 \cdot 10^7$	6.1

Table 2: Performance comparison of FluCaP (denoted as FluCaP) and symbolic LAO* (denoted as LAO*), where the cells n/a denote the fact that a planner did not deliver a solution within the time limit of one hour. NAS and NGS are number of abstract and ground states, respectively.

labelled % justify that the harder the problem is (that is, the more colors it contains), the higher the percentage of runtime spent on normalization. Almost on all test problems, the effort spent on normalization takes three percent of the total runtime on average.

In order to compare the heuristic accuracy, we present in the column labelled NGS the number of ground states which the heuristic assigns non-zero values to. One can see that the heuristics returned by FluCaP and symbolic LAO* have similar accuracy, but FluCaP takes much less time to compute them. This reflects the advantage of the plain first-order abstraction in comparison to the marriage of propositionalization with abstraction using propositional ADDs. In some examples, we gain several orders of magnitude in H.time.

The column labelled Total time presents the time needed to solve a problem. During this time, a planner must execute 30 runs from an initial state to a goal state. A one-hour block is allocated for each problem. We note that, in comparison to FluCaP, the time required by heuristic search in symbolic LAO* (i.e., difference between Total time and H.time) grows considerably faster in the size of the problem. This reflects the potential of employing first-order abstraction instead of abstraction based on propositional ADDs during heuristic search.

The average reward obtained over 30 runs, shown in column Total av. reward, is the planner's evaluation score. The reward value close to 500 (which is the maximum possible reward) simply indicates that a planner found a reasonably good policy. Each time the number of blocks B increases by 1, the running time for symbolic LAO* increases roughly 10 times. Thus, it could not scale to problems having more than seven blocks. This is in contrast to FluCaP which could solve problems of seventeen blocks. We

B	Total av. reward, ≤500	Total time, sec.	H.time, sec.	NAS	NGS × 10^{21}
20	489.0	137.5	56.8	711	1.7
22	487.4	293.8	110.2	976	1.1×10^3
24	492.0	757.3	409.8	1676	1.0×10^6
26	482.8	817.0	117.2	1141	4.6×10^8
28	493.0	2511.3	823.3	2832	8.6×10^{11}
30	491.2	3580.4	1174.0	4290	1.1×10^{15}
32	476.0	3953.8	781.8	2811	7.4×10^{17}
34	475.6	3954.1	939.4	3248	9.6×10^{20}
36	n/a	n/a	n/a	n/a	n/a

Table 3: Performance of FluCaP on larger instances of one-color Blocksworld problems, where the cells n/a denote the fact that a planner did not deliver a solution within the time limit.

note that the number of colors C in a problem affects the efficiency of an abstraction technique. In FluCaP, as C decreases, the abstraction rate increases which, in turn, is reflected by the dramatic decrease in runtime. The opposite holds for symbolic LAO*.

In addition, we compare FluCaP with two variants. The first one, denoted as FOVI, performs no heuristic search at all, but rather, employs FOVI to compute the ε-optimal total value function from which a policy is extracted. The second one, denoted as FluCaP$^-$, performs 'trivial' heuristic search starting with an initial value function as an admissible heuristic. As expected, FluCaP that combines heuristic search and FOVI demonstrates an advantage over plain FOVI and trivial heuristic search. These results illustrate the significance of heuristic search in general (FluCaP vs. FOVI) and the importance of heuristic accuracy, in particular (FluCaP vs. FluCaP$^-$). FOVI and FluCaP$^-$ do not scale to problems with more than seven blocks.

Table 3 presents the performance results of FluCaP on larger instances of one-color BW problems with the number of blocks varying from twenty to thirty four. We believe that FluCaP does not scale to problems of larger size because the implementation is not yet well optimized. In general, we believe that the FluCaP system should not be as sensitive to the size of a problem as propositional planners are.

Our experiments were targeted at the one-color problems only because they are, on the one hand, the simplest ones for us and, on the other hand, the bottleneck for propositional planners. The structure of one-color problems allows us to apply first-order state abstraction in its full power. For example, for a 34-blocks problem FluCaP operates on about 3.3 thousand abstract states that explode to 9.6×10^{41} individual states after propositionalization. A propositional planner must be highly optimized in order to cope with this non-trivial state space.

We note that additional colors in larger instances (more than 20 blocks) of BW problems cause dramatic increase in computational time, so we consider these problems as being unsolved. One should also observe that the number of abstract states NAS increases with the number of blocks non-monotonically because the problems are generated randomly. For example, the 30-blocks problem happens to be harder than the 34-blocks one. Finally, we note that all results that appear in Tables 2 and 3 were obtained by using the new version of the evaluation software that does not rely on propositionalization in contrast to the initial version that was used during the competition.

The competition domains and results are available in [YLWA05].

6 Domain-independent Methods for θ-subsumption

Given two fluent terms Z_1 and Z_2 under singularity condition, Z_1 θ-subsumes Z_2, written $Z_1 \vdash_{\theta}^{AC1} Z_2$, iff there exists an allowed substitution θ such that $(Z_1 \circ U)\theta =_{AC1} Z_2$.

Initially, θ-subsumption was defined on clauses. Given two clauses C and D, C θ-subsumes D iff there exists a substitution θ such that $C\theta \subseteq D$ [Rob65]. In general, θ-subsumption is NP-complete [KN86]. In the domain-dependent implementation of the LIFT-UP approach, that was described in the previous section, we have employed domain-driven optimization techniques that have allowed to reduce the complexity of θ-subsumption. This section is devoted to the efficient domain-independent solution methods for θ-subsumption which cope with its NP-completeness.

One approach to cope with the NP-completeness of θ-subsumption is deterministic subsumption. A state is said to be determinate if there is an ordering of fluents, such that in each step there is a fluent which has exactly one match that is consistent with the previously matched fluents [KL94]. However, in practice, there may be only few fluents, or none at all, that can be matched deterministically. Recently, in [SHW96], it was developed another approach, which we refer to as literal context, LitCon, for short, to cope with the complexity of θ-subsumption. The authors propose to reduce the number of matching candidates for each fluent by using the contextual information. The method is based on the idea that fluents may only be matched to those fluents that possess the same relations up to an arbitrary depth in a clause. As a result, a certain superset of determinate states can be tested for subsumption in polynomial time.

Unfortunately, as it was shown in [KRS06], LitCon does not scale very well up to large depth. Because in some planning problems, the size of state descriptions can be relatively large, it might be necessary to compute the contextual information for large values of the depth parameter. Therefore, we are strongly interested in a technique that scales better than LitCon. In this section, we present an approach, referred to as object context, ObjCon, for short, which demonstrates better computational behaviour than LitCon. Based on the idea of ObjCon, we develop a new θ-subsumption algorithm and compare it with the LitCon-based approach.

6.1 Object Context

In general, a fluent f in a state Z_1 can be matched with several fluents in a state Z_2, that are referred to as matching candidates of f. LitCon is based on the idea that fluents in Z_1 can be only matched to those fluents in Z_2, the context of which include the context of the fluents in Z_1 [SHW96]. The context is given by occurrences of identical objects (variables $\mathit{Vars}(Z)$ and constants $\mathit{Const}(Z)$) or chains of such occurrences and is defined up to some fixed depth. In effect, matching candidates that do not meet the above context condition can be effortlessly pruned. In most cases, such pruning results in deterministic subsumption, thereby considerably extending the tractable class of states.

The computation of the context itself is dramatically affected by the depth parameter: The larger the depth is, the longer the chains of objects' occurrences are, and thus, more effort should be devoted to build them. Unfortunately, LitCon does not scale very well up to large depth [KRS06]. For example, consider a state

$$Z = on(X,Y) \circ on(Y, table) \circ r(X) \circ b(Y) \circ h(X) \circ h(Y) \circ w(X) \circ l(Y)$$

that can be informally read as: A block X is on the block Y which is on the table, and both blocks enjoy various properties, like color (red r or blue b) or weight (heavy h or light l), they can be wet w. Z contains eight fluents and only three objects. In LitCon, the context should be computed for each of eight fluents in order to keep track of all occurrences of identical objects. What if we were to compute the context for each object instead? In our running example, we would need to perform computations only three times, in this case.

Herein, we propose a more efficient approach, referred to as ObjCon, for computing the contextual information and incorporate it into a new context-based θ-subsumption algorithm. More formally, we build the object occurrence graph $\mathcal{G}_Z = (V, E, \ell)$ for a state Z, where vertices are objects of Z, denoted as $Obj(Z)$, and edges $E = \{(o_1, \pi_1, f, \pi_2, o_2) | Z$ contains $f(t_1, \ldots, t_n)$ and $o_1 = t_{\pi_1}$ and $o_2 = t_{\pi_2}\}$ with $o_1, o_2 \in Obj(Z)$, $f(t_1, \ldots, t_n)$ being a fluent and π_1, π_2 being positions of objects o_1, o_2 in f. The labeling function $\ell(o) = \{f | Z \text{ contains } f(o)\}$ associates each object o with a unary fluent name f this object belongs to. The object occurrence graph for the state Z from our running example will contain three vertices X, Y and $table$ with labels $\{r, h, w\}$, $\{b, h, l\}$ and $\{\}$, resp., and two edges $(X, 1, on, 2, Y)$ and $(Y, 1, on, 2, table)$.

The object context ObjCon(o, Z, d) of depth $d > 0$ is defined for each object o of a state Z as a chain of labels: $\ell(o) \xrightarrow{\pi_1^1 \cdot f^1 \cdot \pi_2^1} \ell(o_1) \xrightarrow{\pi_1^2 \cdot f^2 \cdot \pi_2^2} \ldots \xrightarrow{\pi_1^d \cdot f^d \cdot \pi_2^d} \ell(o_d) \in$ ObjCon(o, Z, d) *iff* $o \xrightarrow{\pi_1^1 \cdot f^1 \cdot \pi_2^1} o_1 \xrightarrow{\pi_1^2 \cdot f^2 \cdot \pi_2^2} \ldots \xrightarrow{\pi_1^d \cdot f^d \cdot \pi_2^d} o_d$ is a path in $\mathcal{G}_Z$ of length d starting at o. In our running example, ObjCon$(X, Z, 1)$ of depth 1 of the variable X in Z contains one chain $\{\{r, h, w\} \xrightarrow{1 \cdot on \cdot 2} \{b, h, l\}\}$.

Following the ideas of [SHW96], we define the embedding of object contexts for states Z_1 and Z_2, which serves as a pruning condition for reducing the space of matching candidates for Z_1 and Z_2. Briefly, let $OC_1 =$ObjCon(o_1, Z_1, d), $OC_2 =$ObjCon(o_2, Z_2, d). Then OC_1 is embedded in OC_2, written $OC_1 \preccurlyeq OC_2$, iff for every chain of labels in OC_1 there exists a chain of labels in OC_2 which preserves the positions of objects in fluents and the labels for each object in OC_1 are included in the respective labels in OC_2 up to the depth d. Finally, if ObjCon$(X, Z_1, d) \not\preccurlyeq$ ObjCon(o, Z_2, d) then there exists no θ such that $(Z_1 \circ U)\mu\theta =_{AC1} Z_2$, where $\mu = \{X \mapsto o\}$ and U is a new variable of sort fluent term. In other words, a variable X in Z_1 cannot be matched against an object o in Z_2 within a globally consistent match, if the variable's context cannot be embedded in the object's context. Therefore, the substitutions that meet the above condition can be effortlessly pruned from the search space. For any context depth $d > 0$, the context inclusion is an additional condition that reduces the number of candidates, and hence there exists more often at most one remaining matching candidate.

Based on the idea of the object context, we describe a new θ-subsumption algorithm in Algorithm 1. Please note that this algorithm provides a complete set of all allowed substitutions which is used later on for determining the set of all possible successors or predecessors of some state with respect to some action. Due to the lack of space, we

Input: Two fluent terms Z_1, Z_2.
Output: A complete set of substitutitons θ such that $Z_1 \vdash_{\theta}^{AC1} Z_2$.

1. Deterministically match as many fluents of Z_1 as possible to fluents of Z_2. Substitute Z_1 with the substitution found. If some fluent of Z_1 does not match any fluent of Z_2, decide $Z_1 \nvdash_{\theta}^{AC1} Z_2$.
2. ObjCon-based deterministically match as many fluents of Z_1 as possible to fluents of Z_2. Substitute Z_1 with the substitution found. If some fluent of Z_1 does not match any fluent of Z_2, decide $Z_1 \nvdash_{\theta}^{AC1} Z_2$.
3. Build the substitution graph (V, E) for Z_1 and Z_2 with nodes $v = (\mu, i) \in V$, where μ is a matching candidate for Z_1 and Z_2, i.e., matches some fluent at position i in Z_1 to some fluent in Z_2 and $i \geq 1$ is referred to as a layer of v. Two nodes (μ_1, i_1) and (μ_2, i_2) are connected with an edge iff $\mu_1\mu_2 = \mu_2\mu_1$ and $i_1 = i_2$. Delete all nodes (μ, i) with $X\mu = o$, for some $X \in \mathit{Vars}(Z_1)$ and $o \in \mathit{Obj}(Z_2)$, and ObjCØn$(X, Z_1, d) \preccurlyeq$ ObjCon(o, Z_2, d) for some d. Find all cliques of size $|Z_1|$ in (V, E). /

Algorithm 1: ObjCon-alltheta.

omit the algorithm for computing all cliques in a substitution graph. However, it can be found in [KRS06].

6.2 Experimental Evaluation

Figure 4 depicts the comparison timing results between the LitCon-based subsumption reasoner, referred to as AllTheta, and its ObjCon-based opponent, referred to as FluCaP. The results were obtained using RedHat Linux running on a 2.4GHz Pentium IV machine with 2GB of RAM.

We demonstrate the advantages of exploiting the object-based context information on problems that stem from the colored Blocksworld and Pipesworld planning scenarios. The Pipesworld domain models the flow of oil-derivative liquids through pipeline segments connecting areas, and is inspired by applications in the oil industry. Liquids are modeled as batches of a certain unit size. A segment must always contain a certain number of batches (i.e., it must always be full). Batches can be pushed into pipelines from either side, leading to the batch at the opposite end "falling" into the incident area. Batches have associated product types, and batches of certain types may never be adjacent to each other in a pipeline. Moreover, areas may never have constraints on how many batches of a certain product type they can hold.

For each problem, there have been done 1000 subsumption tests. The time limit of 100 minutes has been allocated. The results show that FluCaP scales better than AllTheta. It is best to observe on the problems of forteen-, twenty-, and thirty-blocks. As empirical results demonstrate, the optimal value of the depth parameter for Blocksworld and Pipesworld is four.

The main reason for the computational gain of FluCaP is that it is less sensitive to the growth of the depth parameter. Under the condition that the number of objects in a state is strictly less than the number of fluents and other parameters are fixed, the amount of object-based context information is strictly less than the amount of the literal-based context information. Moreover, on the Pipesworld problems, FluCaP requires two orders of magnitude less time than AllTheta.

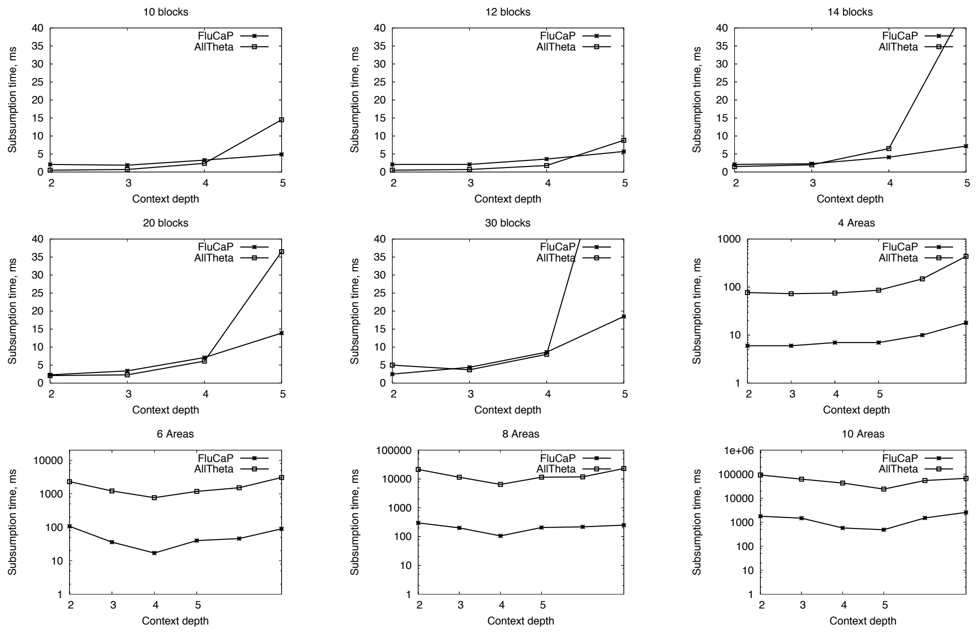


Figure 4: Comparison timing results for FluCaP and AllTheta. The results present the average time needed for one subsumption test. Please note that the plots for Pipesworld are shown in logscale. Therefore small differences in the plot may indicate a substantial difference on runtimes.

7 Related Work

We follow the symbolic dynamic programming (SDP) approach within Situation Calculus (SC) of [BRP01] in using first-order state abstraction for FOMDPs. In the course of first-order value iteration, a state space may contain redundant abstract states that dramatically affect the algorithm's efficiency. In order to achieve computational savings, normalization must be performed to remove this redundancy. However, in the original work by [BRP01] this was done by hand. To the best of our knowledge, the preliminary implementation of the SDP approach within SC uses human-provided rewrite rules for logical simplification. In contrast, [HS04] have developed an automated normalization procedure for FOVI brings the computational gain of several orders of magnitude. Another crucial difference is that our algorithm uses heuristic search to limit the number of states for which a policy is computed.

The ReBel algorithm by [KvOdR04] relates to LIFT-UP in that it also uses a representation language that is simpler than Situation Calculus. This feature makes the state space normalization computationally feasible.

All the above algorithms can be classified as deductive approaches to solving FOMDPs. They can be characterized by the following features: (1) they are model-based, (2) they aim at exact solutions, and (3) logical reasoning methods are used to compute abstractions. We should note that FOVI aims at exact solution for a FOMDP, whereas LIFT-UP, due to the heuristic search that avoids evaluating all states, seeks for an approximate solution. Therefore, it would be more appropriate to classify LIFT-UP as an approximate deductive approach to FOMDPs.

In another vein, there is some research on developing inductive approaches to solving FOMDPs, e.g., by [FYG03]. The authors propose the approximate policy iteration (API) algorithm, where they replace the use of cost-function approximations as policy representations in API with direct, compact state-action mappings, and use a standard relational learner to learn these mappings. A recent approach by [GT04] proposes an inductive policy construction algorithm that strikes a middle-ground between deductive and inductive techniques.

8 Conclusions

We have proposed a new approach that combines heuristic search and first-order state abstraction for solving first-order MDPs more efficiently. In contrast to existing systems, which start with propositionalizing the decision problem at the outset of any solution attempt, we perform lifted reasoning on the first-order structure of an MDP directly. However, there is plenty remaining to be done. For example, we are interested in the question of to what extent the optimization techniques applied in modern propositional planners can be combined with first-order state abstraction.

Acknowledgements

We thank anonymous reviewers for their comments on the previous versions of this paper. We also thank Zhengzhu Feng for fruitful discussions and for providing us with the executable of the symbolic LAO* planner. Olga Skvortsova was supported by a grant within the Graduate Programme GRK 334 "Specification of discrete processes and systems of processes by operational models and logics" under auspices of the Deutsche Forschungsgemeinschaft (DFG).

References

[BBS95] A. G. Barto, S. J. Bradtke, and S. P Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2):81–138, 1995.

[BDH99] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[Bel57] R. E. Bellman. *Dynamic programming.* Princeton University Press, Princeton, NJ, USA, 1957.

[BRP01] C. Boutilier, R. Reiter, and B. Price. Symbolic Dynamic Programming for First-Order MDPs. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Conference on Artificial Intelligence (IJCAI'2001)*, pages 690–700. Morgan Kaufmann, 2001.

[DKKN95] T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76:35–74, 1995.

[FH02] Z. Feng and E. Hansen. Symbolic heuristic search for factored Markov Decision Processes. In R. Dechter, M. Kearns, and R. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'2002)*, pages 455–460, Edmonton, Canada, 2002. AAAI Press.

[FYG03] A. Fern, S. Yoon, and R. Givan. Approximate policy iteration with a policy language bias. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Proceedings of the Seventeenth Annual Conference on Neural Information Processing Systems (NIPS'2003)*, Vancouver, Canada, 2003. MIT Press.

[GT04] C. Gretton and S. Thiebaux. Exploiting first-order regression in inductive policy selection. In M. Chickering and J. Halpern, editors, *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence (UAI'2004)*, Banff, Canada, July 2004. Morgan Kaufmann.

[HS90] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990.

[HS04] S. Hölldobler and O. Skvortsova. A Logic-Based Approach to Dynamic Programming. In *Proceedings of the Workshop on "Learning and Planning in Markov Processes–Advances and Challenges" at the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, pages 31–36, San Jose, CA, July 2004. AAAI Press.

[HZ01] E. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129:35–62, 2001.

[KL94] J.-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In *Proceedings of the Eleventh International Conference on MAchine Learning*, pages 130–138, 1994.

[KN86] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In J.H.Siekman, editor, *Proceedings of the Conference on Automated Deduction*, pages 489–495. Springer, Berlin, 1986. Lecture Notes in Computer Science 230.

[KRS06] E. Karabaev, G. Rammé, and O. Skvortsova. Efficient symbolic reasoning for first-order MDPs. In *Proceedings of the Workshop on "Planning, Learning and Monitoring with Uncertainty and Dynamic Worlds" at the Seventeenth European Conference on Artificial Intelligence (ECAI'2006)*, Riva del Garda, Italy, 2006. To appear.

[KvOdR04] K. Kersting, M. van Otterlo, and L. de Raedt. Bellman goes relational. In C. E. Brodley, editor, *Proceedings of the Twenty-First International Conference in Machine Learning (ICML'2004)*, pages 465–472, Banff, Canada, July 2004. ACM.

[Put94] M. L. Puterman. *Markov Decision Processes - Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, NY, 1994.

[Rei91] R. Reiter. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[Rob65] J.A. Robinson. A machine-learning logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[SAHB00] R. St-Aubin, H. Hoey, and C. Boutilier. APRICODD: Approximate policy construction using decision diagrams. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Proceedings of the Fourteenth Annual Conference on Neural Information Processing Systems (NIPS'2000)*, pages 1089–1095, Denver, 2000. MIT Press.

[SHW96] T. Scheffer, R. Herbrich, and F. Wysotzki. Efficient θ-subsumption based on graph algorithms. In *Proceedings of the 6th International Workshop on Inductive Logic Programming*, pages 212–228, Berlin, August 1996. volume 1314 of LNAI.

[Thi98] M. Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, 2(3-4):179–192, 1998.

[YLWA05] H.L.S. Younes, M.L. Littman, D. Weissman, and J. Asmuth. The first probabilistic track of the International Planning Competition. *Journal of Artificial Intelligence Research*, 24:851–887, 2005.

Multiple Preprocessing for Systematic SAT Solvers

Anbulagan[1], John Slaney[1,2]
[1]Logic and Computation Program, National ICT Australia Ltd.
[2]Computer Sciences Laboratory, Australian National University
{anbulagan, john.slaney}@nicta.com.au

Abstract

High-performance SAT solvers based on systematic search generally use either conflict driven clause learning (CDCL) or lookahead techniques to gain efficiency. Both styles of reasoning can gain from a preprocessing phase in which some form of deduction is used to simplify the problem. In this paper we undertake an empirical examination of the effects of several recently proposed preprocessors on both CDCL and lookahead-based SAT solvers. One finding is that the use of multiple preprocessors one after the other can be much more effective than using any one of them alone, but that the order in which they are applied is significant. We intend our results to be particularly useful to those implementing new preprocessors and solvers.

1 Introduction

In the last decade, the propositional satisfiability (SAT) has become one of the most interesting research problems within artificial intelligence (AI). This tendency can be seen through the development of a number of powerful SAT solvers, based on either systematic search or stochastic local search (SLS), for solving various hard combinatorial search problems such as automatic deduction, hardware and software verification, planning, scheduling, and FPGA routing.

The power of contemporary systematic SAT solvers derives not only from the underlying Davis-Putnam-Logemann-Loveland (DPLL) algorithm but also from enhancements aimed at increasing the amount of unit propagation, improving the choices of variables for splitting or making backtracking more intelligent. Two of the most important such enhancements are conflict driven clause learning (CDCL), made practicable on a large scale by the watched literal technique, and one-step lookahead. These two tend to exclude each other: the most successful solvers generally incorporate one or the other but not both. The benefits they bring are rather different too, as is clear from the results of recent SAT competitions. For problems in the "industrial" category, CDCL, as implemented in MiniSat [ES03, SE05], siege [Rya04] and zChaff [MMZ$^+$01, ZMMM01] is currently the method of choice. On random problems, however, lookahead-based solvers such as Dew_Satz [AS05], Kcnfs [DD01] and March_dl [HvM06] perform better.

Lookahead, of course, is expensive at every choice node, while clause learning is expensive only at backtrack points. Since half of the nodes (plus one) in any binary tree are leaves, this difference is significant for lookahead-based solvers which process nodes relatively slowly and gain, if at all, by reducing the search tree size. Looking ahead is an investment in at-node processing which can pay off only if it results in more informed choices with an impact on the total number of nodes visited. Where learnt clauses prune the tree at least as effectively as complex choice heuristics, CDCL must win. This seems to be the case in many classes of highly structured problems such as the "industrial" ones in the SAT competitions. We have no clearer definition than anyone else of "structure", but are interested to find ways in which lookahead-based solvers might detect and exploit it as well as the clause learners do.

A noteworthy feature of many recent systems is a preprocessing phase, often using inference by some variant of resolution, to transform problems prior to the search. One suggestion we wish to make and explore is that such transformations may help lookahead-based solvers to discover useful structure. That is, much of the reasoning done by nogood inference might be done cheaply "up-front", provided that the subsequent variable choice heuristics are good enough to exploit it. In what follows, we are therefore concerned mainly with the effects of preprocessing, including those of using multiple preprocessors in series, on the performance of a lookahead-based solver Dew_Satz on problems where it does poorly in comparison with a clause learning solver (MiniSat). We also include some results and remarks on benefits to be gained in the opposite direction, where MiniSat is helped by preprocessing to attack problems to which Dew_Satz is more suited.

In this paper we propose a multiple preprocessing technique to boost the performance of systematic SAT solvers. The motivation for applying multiple preprocessors prior to the systematic search process is clear: each preprocessor uses a different strategy in objective to simplify clause sets derived from real-world problems that exhibit a great deal structure such as symmetries, variable dependencies, clustering, and the like. Our initial observation showed that each strategy works well for simplifying the structure of some problems, at most of the time, from hard to easy. When a problem exhibits different kinds of structure, then a single preprocessor has difficulty to simplify the structures. In this case, we need to run multiple preprocessors one after the other.

We report performance statistics for the two solvers, Dew_Satz and MiniSat, with and without combinations of five (and for one problem set, six) of the best contemporary SAT preprocessors when solving parity, planning, bounded model checking and FPGA routing benchmark problems from SATLIB and the recent SAT competitions. One finding is that the use of multiple preprocessors one after the other can be much more effective than using any one of them alone, but that the order in which they are applied is significant. We intend our results to be particularly useful to those implementing new preprocessors and solvers.

The rest of the paper is organized as follows: section 2 addresses the related work. In sections 3 and 4, we briefly describe the preprocessors and solvers examined in our study. The main part of the paper consists of experimental results, and we conclude with a few remarks and suggestions.

2 Related Work

Resolution-based SAT preprocessors for CNF formula simplification have a dramatic impact on the performance of even the most efficient SAT solvers on many benchmark problems [LMS01]. The simplest preprocessor consists of just computing length-bounded resolvents and deleting duplicate and subsumed clauses, as well as tautologies and any duplicate literals in a clause.

There are two most directly related works. The first one is that of Anbulagan *et al.* [APSS06] which examined the integration of five resolution-based preprocessors alone or the combination of them with stochastic local search (SLS) solvers. Their experimental results show that SLS solvers benefit the present of resolution-based preprocessing and multiple preprocessing techniques. And the second one is that of Lynce and Marques-Silva [LMS01]. They only evaluated empirically the impact of some preprocessors developed before 2001 including 3-Resolution, without considering multiple preprocessing, on the performance of systematic SAT solvers. In recent years, many other preprocessors, which are sophisticated, have been applied to modern propositional reasoners. Among them are 2-SIMPLIFY [Bra01], the preprocessor in Lsat [OGMS02] for recovering and exploiting Boolean gates, HyPre [Bac02, BW04], Shatter [ASM03] for dealing with symmetry structure, NiVER [SP05] and SatELite [EB05]. We consider some of these preprocessors plus 3-Resolution in our study.

3 SAT Preprocessors

We describe briefly the six SAT preprocessors used in the experiments. The first five are all based on resolution and its variants such as hyper-resolution. Resolution [Qui55, DP60, Rob65] itself is widely used as a rule of inference in first order automated deduction, where the clauses tend to be few in number and contain few literals, and where the reasoning is primarily driven by unification. As a procedure for propositional reasoning, however, resolution is rarely used on its own because in practice it has not been found to lead to efficient algorithms. The sixth preprocessor is a special-purpose tool for symmetry detection, which is important for one problem class in the experiments.

3.1 3-Resolution

k-Resolution is just saturation under resolution with the restriction that the parent clauses are of length at most k. The special cases of 2-Resolution and 3-Resolution are of most interest. 3-Resolution has been used in a number of SAT solvers, notably Satz [LA97] and the SLS solver R+AdaptNovelty$^+$ [APSS05] which won the satisfiable random problem category in the SAT2005 competition. Since it is the preprocessor already used by Satz, we expect it to work well with Dew_Satz.

3.2 2-SIMPLIFY

2-SIMPLIFY [Bra01] constructs an implication graph from all binary clauses in the problem. Where there is an implication chain from a literal X to $\overline{X}$, $\overline{X}$ can be deduced as a unit which can be propagated. The method also collapses strongly connected components, propagates *shared implications*, or literals implied in the graph by every literal in a clause, and removes some redundant binary clauses. Experimental results [Bra01, Bra04] show that systematic search benefits markedly from 2-SIMPLIFY on a wide range of problems.

3.3 HyPre

HyPre [BW04] also reasons with binary clauses, but incorporates full hyper-resolution, making it more powerful than 2-SIMPLIFY. In addition, unit reduction and equality reduction are incrementally applied to infer more binary clauses. It can be costly in terms of time, but since it is based explicitly on hyper-resolution it avoids the space explosion of computing a full transitive closure. HyPre has been used in the SAT solver, 2CLS+EQ [Bac02], and we consider it a very promising addition to many other solvers. It is generally useful for exploiting implicational structure in large problems.

3.4 NiVER

Variable Elimination Resolution (VER) is an ancient inference method consisting of performing all resolutions on a chosen variable and then deleting all clauses in which that variable occurs, leaving just the resolvents. It is easy to see that this is a complete decision procedure for SAT problems, and almost as easy to see that it is not practicable because of exponential space complexity. Recently, Subbarayan and Pradhan [SP05] proposed NiVER (Non increasing VER) which restricts the variable elimination to the case in which there is no increase in the number of literals after elimination. This shows promise as a SAT preprocessor, improving the performance of a number of solvers [SP05].

3.5 SatELite

Eén and Biere [EB05] proposed the SatELite preprocessor, which extends NiVER with a rule of Variable Elimination by Substitution. Several additions including subsumption detection and improved data structures further improved performance in both space and time. SatELite was combined with MiniSat to form SatELiteGTI, the system which dominated the SAT2005 competition on the crafted and industrial problem categories. Since we use MiniSat for our experiments, it is obvious that SatELite should be one of the preprocessors we consider.

3.6 Shatter

It is clear that eliminating symmetries is essential to solving realistic instances of many problems. None of the resolution-based preprocessors does this, so for problems that involve a high degree of symmetry we added Shatter [AMS03] which detects symmetries and adds symmetry-breaking clauses. These always increase the size of the clause set and for satisfiable problems they remove some of the solutions, but they typically make the problem easier by pruning away isomorphic copies of parts of the search space.

4 SAT Solvers

As noted in Section 1, we concentrate on just two solvers: MiniSat, which relies on clause learning, and Dew_Satz, which uses lookahead.

4.1 MiniSat

Sörensson and Eén [ES03, SE05] released the MiniSat solver in 2005. Its design is based on Chaff, particularly in that it learns nogoods or "conflict clauses" and accesses them during the search by means of two watched literals in each clause. MiniSat is quite small (a few hundred lines of code) and easy to use either alone or as a module of a larger system. Its speed in comparison with similar solvers such as zChaff comes from a series of innovations of which the most important are an activity-decay schedule which proceeds by frequent small reductions rather than occasional large ones, and an inference rule for reducing the size of conflict clauses by introducing a restricted subsumption test. The cited paper contains a brief but informative description of these ideas.

4.2 Dew_Satz

The solver Dew_Satz [AS05] is a recent version of the Satz solver [LA97]. Like its parent Satz, it gains efficiency by a restricted one-step lookahead scheme which rates some of the neighbouring variables every time a choice must be made for branching purposes. Its lookahead is more sophisticated than the original one of Satz, adding a DEW (dynamic equality weighting) heuristic to deal with equalities. This enables the variable selection process to avoid duplicating the work of weighting variables detected to be equivalent to those already examined. Thus, while the solver has no special inference mechanism for propositional equalities, it does deal tolerably well with problems containing them.

5 Experimental Results

We present results on four benchmark problem sets chosen to present challenges for one or other or both of the SAT solvers. The experiments were conducted on a cluster of 16 AMD Athlon 64 processors running at 2 GHz with 2 GB of RAM. Ptime in the tables represents preprocessing time, while Stime represents solvers runtime without including Ptime. The timebound of Stime is 15,000 seconds per problem instance. It is worth noting that in our study the results of SatELiteGTI, the solver which dominated the SAT2005 competition on the crafted and industrial problem categories, are represented by the results of SatELite+MiniSat.

5.1 The 32-bit Parity Problem

The 32-bit parity problem was listed by Selman *et al.* [SKM97] as one of ten challenges for research on satisfiability testing. The ten instances of the problem are satisfiable. The first response to this challenge was by Warners and van Maaren [WvM98] who solved the `par32-*-c` problem (5 instances) using a special-purpose preprocessor to deal with equivalency conditions. Two years later, Li [Li00] solved the ten `par32*` instances by enhancing Satz's search process with equivalency reasoning. Ostrowski *et*

Instance	Prep.	#Vars/#Cls/#Lits	Ptime	Dew_Satz		MiniSat	
				Stime	#BackT	Stime	#Conflict
par32-1	Orig	3176/10227/27501	n/a	>15,000	n/a	>15,000	n/a
	3Res	2418/7463/19750	0.08	12,873	17,335,530	>15,000	n/a
	Hyp+3Res	1313/6193/17203	0.50	9,513	17,391,333	>15,000	n/a
	Niv+3Res	1315/5948/16707	0.46	6,858	13,476,105	>15,000	n/a
	3Res+Hyp	1313/5495/15810	0.11	9,655	17,335,492	>15,000	n/a
	Sat+3Res	849/5245/18660	0.37	14,729	34,569,968	>15,000	n/a
par32-2	Orig	3176/10253/27405	n/a	>15,000	n/a	5,364	9,125,821
	3Res	2392/7387/19550	0.08	5,171	9,341,185	6,205	10,492,612
	Hyp+3Res	1301/5975/16719	0.36	3,831	8,186,883	>15,000	n/a
	Niv+3Res	1303/5730/16223	0.29	1,518	3,889,345	>15,000	n/a
par32-3	Orig	3176/10297/27581	n/a	>15,000	n/a	>15,000	n/a
	3Res	2395/7437/19738	0.07	6,124	9,711,576	>15,000	n/a
	Hyp+3Res	1323/5961/16779	0.23	3,673	9,708,520	>15,000	n/a
	Niv+3Res	1325/5716/16283	0.22	4,470	9,710,552	>15,000	n/a
	Sat+3Res	848/5284/18878	0.37	3,647	2,206,369	>15,000	n/a
par32-4	Orig	3176/10313/27645	n/a	>15,000	n/a	>15,000	n/a
	3Res	2385/7433/19762	0.08	10,425	10,036,154	>15,000	n/a
	Sat	849/5160/18581	0.21	12,820	18,230,746	>15,000	n/a
	Hyp+3Res	1331/6055/16999	0.36	9,001	17,712,997	>15,000	n/a
	3Res+Hyp	1331/5567/16026	0.11	5,741	10,036,146	>15,000	n/a
	Niv+3Res	1333/5810/16503	0.34	6,099	10,036,154	>15,000	n/a
	3Res+Niv	1290/5297/15481	0.10	14,003	25,092,756	>15,000	n/a
	3Res+Sat	850/5286/18958	0.35	3,552	7,744,986	>15,000	n/a
	Sat+3Res	849/5333/19052	0.38	3,563	7,744,986	>15,000	n/a
	Sat+2Sim	848/5154/18565	0.26	12,862	18,230,746	>15,000	n/a
par32-5	Orig	3176/10325/27693	n/a	>15,000	n/a	>15,000	n/a
	Niv	1978/7864/22535	0.03	10,651	27,165,469	>15,000	n/a
par32-1-c	Orig	1315/5254/15390	n/a	>15,000	n/a	>15,000	n/a
	3Res	1315/5957/16738	0.35	11,068	25,920,943	>15,000	n/a
	Hyp+3Res	1313/6193/17203	0.48	7,419	8,931,149	>15,000	n/a
par32-2-c	Orig	1303/5206/15246	n/a	>15,000	n/a	>15,000	n/a
	3Res	1303/5739/16254	0.23	428	345,680	>15,000	n/a
	Hyp+3Res	1301/5975/16719	0.32	7,402	8,166,758	>15,000	n/a
par32-3-c	Orig	1325/5294/15510	n/a	>15,000	n/a	>15,000	n/a
	3Res	1325/5725/16314	0.15	4,482	9,462,205	>15,000	n/a
	Hyp	1323/5589/16094	0.04	11,745	19,947,965	>15,000	n/a
	Hyp+3Res	1323/5961/16779	0.22	4,375	9,462,245	>15,000	n/a
	Niv+3Res	1321/5708/16266	0.24	7,361	16,265,438	>15,000	n/a
	Sat+3Res	802/5335/19335	0.33	5,407	7,280,963	>15,000	n/a
par32-4-c	Orig	1333/5326/15606	n/a	>15,000	n/a	>15,000	n/a
	3Res	1333/5819/16534	0.24	7,097	8,440,212	>15,000	n/a
	Hyp+3Res	1331/6055/16999	0.32	5,175	9,669,012	>15,000	n/a
	Niv+3Res	1329/5802/16486	0.55	10,495	21,738,376	>15,000	n/a
	Sat+3Res	806/5357/19443	0.32	10,110	8,013,977	>15,000	n/a
par32-5-c	Orig	1339/5350/15678	n/a	10,949	22,878,571	>15,000	n/a
	Niv+3Res	1335/5728/16362	0.28	>15,000	n/a	7,363	16,189,524

Table 1: Dew_Satz and MiniSat performance, before and after preprocessing, on par32 problem.

al. [OGMS02], solved the problems with Lsat, which performs a preprocessing step to recover and exploit the logical gates of a given CNF formula and then applies DPLL with a Jeroslow-Wang branching rule. The challenge has now been met convincingly by Heule *et al.* [HvM04] with their March_eq solver, which combines equivalency reasoning in a preprocessor with a lookahead-based DPLL and which solves all of the `par32*` instances in seconds. Dew_Satz is one of the few solvers to have solved any instances of the 32-bit parity problem without special-purpose equivalency reasoning [AS05].

Table 1 shows the results of running the lookahead-based solver Dew_Satz and the CDCL-based solver MiniSat on the ten `par32` instances, with and without preprocessing. As preprocessors we used 3-Resolution, HyPre, NiVER and SatELite alone and followed by 3-Resolution for the last three. We eliminated 2-SIMPLIFY from this test as it aborted the resolution process of the first five `par32*` instances presented in the Table 1. We experimented also with all combination of two preprocessors for the problems `par32-1` and `par32-4`. Where lines are omitted from the table (e.g. there is no line for HyPre on `par32-1` and for SatELite+3-Resolution on `par32-2`), this is because no single solver produced a solution for those simplified instances.

It is evident from the table that these problems are seriously hard for both solvers. Even with preprocessing, MiniSat times out on all of them except for `par32-2` and `par32-5-c`. Curiously, on `par32-2` instance, preprocessing with 3-Resolution makes its performance degrade a little. This is not a uniform effect: Table 4 below shows examples in which MiniSat benefits markedly from 3-Resolution. Without preprocessing, Dew_Satz times out on nine of ten `par32` instances, but in every case except `par32-5` and `par32-5-c` 3-Resolution suffices to help it find a solution, and running multiple preprocessors improves its performance.

In general, Table 1 shows that multiple preprocessing contributes significantly to enhance the performance of Dew_Satz and the preprocessor 3-Resolution dominates the contribution through either single or multiple preprocessing.

5.2 A Planning Benchmark Problem

The `ferry` planning benchmark problems, taken from SAT2005 competition, are all easy for MiniSat, which solves all of them in about one second without needing preprocessors. Dew_Satz, however, is challenged by them. The problems are satisfiable. We show the Dew_Satz and MiniSat results on the problems in Table 2. Clearly the original problems contain some structure that CDCL is able to exploit but which is uncovered by one-step lookahead. It is therefore interesting to see which kinds of reasoning carried out in a preprocessing phase are able to make that same structure available to Dew_Satz. Most strikingly, reasoning with binary clauses in the manner of the 2-SIMPLIFY preprocessor reduces runtimes by upwards of four orders of magnitude in some cases. HyPre, NiVER and SatELite, especially HyPre, are also effective on these planning problems. In most cases the number of backtracks reduces from million to less than 100 or even zero for `ferry8_v01a`, `ferry9_v01a`, and `ferry10_ks99a` instances which means that the input formula is solved at the root node of the search tree.

Instance	Prep.	#Vars/#Cls/#Lits	Ptime	Dew_Satz		MiniSat	
				Stime	#BackT	Stime	#Conflict
ferry7_ks99i	Orig	1946/22336/45706	n/a	2,828	10,764,261	0.13	4,266
	3Res	1930/22289/45621	0.09	>15,000	n/a	0.11	3,707
	Hyp	1881/32855/66732	0.21	1,672	1,204,321	0.03	417
	Niv	1543/21904/45243	0.01	>15,000	n/a	0.10	3,469
	Sat	1286/21601/50644	0.33	>15,000	n/a	0.07	2,763
	Sat+2Sim	1279/56597/120318	0.49	0.41	28	0.05	1,096
ferry7_v01i	Orig	1329/21688/50617	n/a	>15,000	n/a	0.05	1,858
	3Res	1329/21681/50505	0.14	>15,000	n/a	0.05	1,858
	Sat	1286/21609/50803	0.17	>15,000	n/a	0.18	6,309
	Sat+2Sim+3Res	1286/64472/136299	0.95	4.28	824	0.05	1,018
	Sat+2Sim+Hyp+3Res	1272/62208/131357	1.26	3.30	580	0.10	2,398
ferry8_ks99a	Orig	1259/15259/31167	n/a	574	1,869,995	0.01	0
	3Res	1241/15206/31071	0.08	654	2,074,794	0.01	0
	Sat	813/14720/34687	0.24	810	1,040,528	0.01	381
	Sat+2Sim	813/35008/75263	0.35	0.11	4	0.02	295
ferry8_ks99i	Orig	2547/32525/66425	n/a	>15,000	n/a	0.22	6,615
	3Res	2529/32472/66329	0.12	>15,000	n/a	0.14	3,495
	Hyp	2473/48120/97601	0.32	>15,000	n/a	0.07	1,030
	Sat	1696/31589/74007	0.49	>15,000	n/a	0.41	10,551
	Sat+2Sim	1683/83930/178217	0.76	9.38	3,255	0.20	5,105
ferry8_v01a	Orig	854/14819/34624	n/a	13,162	39,153,348	0.01	277
	3Res	854/14811/34480	0.11	>15,000	n/a	0.01	277
	Hyp	846/38141/81268	0.18	6.11	570	0.02	226
	Hyp+Sat	813/38044/81364	0.36	29.66	2,749	0.02	277
	Hyp+Sat+3Res	813/38028/81196	0.70	0.71	15	0.02	277
	Hyp+Sat+2Sim	813/36583/78442	0.50	0.17	0	0.02	233
ferry8_v01i	Orig	1745/31688/73934	n/a	>15,000	n/a	0.55	12,935
	3Res	1745/31680/73790	0.20	>15,000	n/a	0.55	12,935
	Sat	1696/31598/74202	0.25	>15,000	n/a	0.25	7,869
	Sat+2Sim+3Res	1696/96092/202904	1.50	268	68,681	4.06	28,690
ferry9_ks99a	Orig	1598/21427/43693	n/a	>15,000	n/a	0.01	0
	3Res	1578/21368/43586	0.10	>15,000	n/a	0.01	0
	Hyp	1542/29836/60522	0.21	>15,000	n/a	0.02	278
	Niv	1244/21046/43264	0.01	>15,000	n/a	0.01	29
	Sat	1042/20765/48878	0.33	>15,000	n/a	0.02	350
	2Sim	1569/20563/41976	0.02	>15,000	n/a	0.04	1,359
	Hyp+Sat	1056/26902/72553	0.88	33.73	22,929	0.03	609
	Sat+2Sim	1042/50487/108322	0.50	0.18	5	0.03	261
ferry9_v01a	Orig	1088/20878/48771	n/a	>15,000	n/a	0.01	181
	3Res	1088/20869/48591	0.16	>15,000	n/a	0.01	181
	Hyp	1079/55371/117757	0.28	0.42	0	0.03	187
	Hyp+Sat	1042/55256/117861	0.49	70.83	5,080	0.03	181
	Hyp+Sat+2Sim	1042/53394/114135	0.72	0.39	2	0.03	234
ferry10_ks99a	Orig	1977/29041/59135	n/a	>15,000	n/a	0.03	710
	3Res	1955/28976/59017	0.13	>15,000	n/a	0.03	827
	Hyp	1915/40743/82551	0.29	>15,000	n/a	0.04	563
	Niv	1544/28578/58619	0.02	>15,000	n/a	0.01	0
	Sat	1299/28246/66432	0.44	>15,000	n/a	0.03	909
	2Sim	1945/27992/57049	0.05	>15,000	n/a	0.05	1,565
	Sat+2Sim	1299/69894/149728	0.69	0.28	1	0.04	419
	3Res+2Sim+Niv	1793/21099/43369	0.43	0.08	0	0.06	1,278
	Niv+Hyp+2Sim+3Res	1532/24524/50463	0.54	5.19	3,949	0.02	454
ferry10_v01a	Orig	1350/28371/66258	n/a	>15,000	n/a	0.02	191
	3Res	1350/28361/66038	0.23	>15,000	n/a	0.02	191
	Hyp	1340/77030/163576	0.40	4.90	550	0.04	401
	Hyp+Sat+3Res	1299/76874/163442	1.56	1,643	118,635	0.04	343
	Hyp+Sat+2Sim	1299/74615/159134	1.00	1.78	61	0.04	459

Table 2: Dew_Satz and MiniSat performance, before and after preprocessing, on ferry planning problem.

Instance	Prep.	#Vars/#Cls/#Lits	Ptime	Dew_Satz		MiniSat	
				Stime	#BackT	Stime	#Conflict
bmc-ibm-3	Orig	14930/72106/189182	n/a	>15,000	n/a	0.39	1,738
	Hyp	5429/32038/89471	3.72	113	2,327	0.06	379
	Niv	10591/62966/176261	0.47	>15,000	n/a	0.36	2,088
	3Res	11940/56736/148383	0.41	>15,000	n/a	0.37	2,374
	Sat	6486/44239/137653	1.32	>15,000	n/a	0.21	1,744
	Hyp+3Res	5429/30517/79041	3.98	19.67	68	0.04	206
bmc-galileo-8	Orig	58073/294821/767187	n/a	>15,000	n/a	0.37	462
	Hyp	9613/85311/202625	416	67.92	0	0.06	2
	Niv	30788/240141/685499	1.06	>15,000	n/a	0.42	1,148
	3Res	43962/182261/456906	5.91	>15,000	n/a	0.46	1,182
	Sat	20593/135076/414093	6.46	>15,000	n/a	0.19	762
	Hyp+3Res	9613/82572/188966	417	120	0	0.05	2
	Sat+3Res	20561/134793/413048	10.12	19.00	1	0.18	799
bmc-galileo-9	Orig	63623/326999/852078	n/a	>15,000	n/a	0.54	1,186
	Hyp	8802/70198/170970	407	90.50	0	0.06	2
	Niv	33872/267378/763037	1.17	>15,000	n/a	0.42	1,148
	3Res	49400/208310/523628	6.56	>15,000	n/a	0.46	1,060
	Sat	23381/155837/477951	7.47	>15,000	n/a	0.24	1,009
	Hyp+3Res	8802/67042/155884	408	57.57	0	0.04	2
bmc-ibm-10	Orig	59056/323700/854093	n/a	>15,000	n/a	1.77	4,277
	Hyp	2259/10831/29155	47.28	0.30	0	0.01	0
	Niv	40530/285198/797443	1.19	>15,000	n/a	0.90	3,532
	3Res	32377/154730/400447	4.70	>15,000	n/a	1.32	3,276
	Sat	14956/116772/404199	5.53	>15,000	n/a	0.49	2,502
bmc-ibm-11	Orig	32109/150027/394770	n/a	>15,000	n/a	2.01	6,422
	Hyp	7342/40802/106327	13.84	4.73	1	0.05	160
	Niv	22927/130058/365568	0.96	>15,000	n/a	1.33	5,607
	3Res	22709/98066/252495	1.46	>15,000	n/a	2.44	7,875
	Sat	10071/62668/200137	2.58	>15,000	n/a	0.77	5,481
bmc-ibm-12	Orig	39598/194778/515536	n/a	>15,000	n/a	8.41	11,887
	Hyp	12205/87082/228241	91.61	>15,000	n/a	0.74	1,513
	Niv	27813/168440/476976	0.69	>15,000	n/a	4.46	8,702
	3Res	32606/160555/419341	2.77	>15,000	n/a	6.77	10,243
	Sat	15176/109121/364968	4.50	>15,000	n/a	2.37	6,219
	Niv+Hyp+ 3Res	12001/100114/253071	85.81	106	6	0.76	1,937
bmc-ibm-13	Orig	13215/65728/174164	n/a	>15,000	n/a	1.84	8,088
	Hyp	5010/27248/78059	3.16	>15,000	n/a	0.13	1,018
	Niv	9226/57332/161962	0.35	>15,000	n/a	1.72	9,181
	3Res	10426/49594/129998	0.49	>15,000	n/a	13.17	30,687
	Sat	4549/34273/110676	1.27	>15,000	n/a	1.25	9,324
	3Res+Niv+ Hyp+3Res	3529/22589/62633	2.90	1,575	4,662,067	0.03	150
bmc-alpha-25449	Orig	663443/3065529/7845396	n/a	>15,000	n/a	6.64	502
	Sat	12408/76025/247622	129	6.94	7	0.06	1
	Sat+Hyp	9091/61789/203593	566	7.82	2	0.10	109
	Sat+Niv	12356/75709/246367	130	4.48	2	0.06	1
	Sat+3Res	12404/77805/249192	130	8.84	1	0.06	1
	Sat+2Sim	10457/71128/229499	131	6.37	10	0.10	133
bmc-alpha-4408	Orig	1080015/3054591/7395935	n/a	>15,000	n/a	5,409	587,755
	Sat	23657/112343/364874	47.22	>15,000	n/a	1,266	820,043
	Sat+Hyp	13235/88976/263053	56.13	>15,000	n/a	8,753	4,916,981
	Sat+Niv	22983/108603/351369	49.34	>15,000	n/a	2,137	1,294,590
	Sat+3Res	23657/117795/380389	48.18	>15,000	n/a	946	618,853
	Sat+2Sim	17470/129245/375444	51.55	>15,000	n/a	804	561,529
	Sat+2Sim+ 3Res	16837/98726/305057	52.89	>15,000	n/a	571	510,705

Table 3: Dew_Satz and MiniSat performance, before and after preprocessing, on hard BMC instances.

Instance	Prep.	#Vars/#Cls/#Lits	Ptime	Dew_Satz		MiniSat	
				Stime	#BackT	Stime	#Conflict
01-k10	Orig	9275/38802/98468	n/a	18.96	1,472	0.08	313
	Hyp	n/a	1.27	n/a	n/a	n/a	n/a
	Niv	6662/33394/90715	0.16	4.12	366	0.07	327
	3Res	6498/27318/70158	0.24	26.38	2,860	0.07	282
	Sat	3418/19648/62925	0.84	3.46	140	0.03	262
	2Sim	4379/59765/133585	3.41	0.24	1	0.05	135
01-k15	Orig	11524/48585/123966	n/a	>15,000	n/a	0.49	3,743
	3Res+Sat+ Niv+Hyp	3382/25936/79364	4.65	1,420	130,013	0.06	682
	3Res+Hyp+ Niv+3Res	4203/23731/60639	4.33	190	13,449	0.06	430
	3Res+Hyp+ 3Res	4732/24972/63133	4.17	262	19,701	0.04	243
	Hyp	4889/27056/71819	3.94	2,937	178,245	0.06	603
	Niv	8068/41368/113317	0.19	>15,000	n/a	0.49	3,548
	3Res	9403/40059/103137	0.27	>15,000	n/a	0.66	3,783
	Sat	5198/30697/97961	1.13	>15,000	n/a	0.32	3,655
01-k20	Orig	15069/63760/163081	n/a	>15,000	n/a	4.95	16,658
	3Res+Hyp+ Niv+3Res	6382/34846/89807	6.94	513	29,629	0.20	1,261
	Hyp	7323/39150/104635	6.40	>15,000	n/a	0.95	5,182
	Niv	10533/54293/149192	0.25	>15,000	n/a	0.28	2,069
	3Res	12948/55490/142966	0.37	>15,000	n/a	1.58	9,341
	Sat	7179/42837/136537	1.52	>15,000	n/a	1.11	8,705
	2Sim	9370/93921/217635	1.91	>15,000	n/a	0.35	1,977
26-k70	Orig	346561/1752741/4579945	n/a	>15,000	n/a	8,382	2,654,614
	3Res	346561/1756001/4588705	150	21.32	1	1.02	10
	Hyp	243461/1569549/4182061	338	>15,000	n/a	1.22	642
	Niv	155221/1354556/4075072	479	>15,000	n/a	1.07	492
	Sat	132670/1300914/4980854	109	>15,000	n/a	2,325	1,503,271
26-k75	Orig	371091/1877066/4904440	n/a	>15,000	n/a	8,540	2,880,376
	3Res	371091/1880536/4913780	161	22.81	1	1.06	11
	Hyp	260621/1680704/4477966	364	>15,000	n/a	1.20	654
	Niv	166195/1450679/4364543	4.95	>15,000	n/a	1.41	474
	Sat	141870/1392526/5327557	117	>15,000	n/a	3,896	2,067,948
26-k85	Orig	420151/2125716/5553430	n/a	>15,000	n/a	>15,000	n/a
	3Res	420151/2129606/5563930	183	25.43	1	1.21	10
	Hyp	294941/1903014/5069776	417	>15,000	n/a	1.55	747
	Niv	187631/1641901/4941437	5.69	>15,000	n/a	1.37	535
	Sat	160270/1576770/6039510	132	>15,000	n/a	4,472	2,308,225
26-k90	Orig	444681/2250041/5877925	n/a	>15,000	n/a	>15,000	n/a
	Niv+3Res	198605/2208074/6624608	121	13.21	1	1.77	5
	Hyp+3Res	312101/1979389/5187311	600	46.53	1	1.39	5
	3Res	444681/2254141/5889005	195	26.99	1	1.26	10
	Hyp	312101/2014169/5365681	446	>15,000	n/a	1.55	583
	Niv	198605/1738024/5230908	5.91	>15,000	n/a	1.38	429
	Sat	169470/1669436/6402318	140	>15,000	n/a	8,240	3,311,629

Table 4: Dew_Satz and MiniSat performance, before and after preprocessing, on SAT2005 IBM-FV-* instances.

5.3 Bounded Model Checking Problems

Another domain providing benchmark problem sets which appear to be easy for MiniSat but sometimes hard for Dew_Satz is bounded model checking. In Table 3 we report results on five of eleven `BMC-IBM` problems, two `BMC-galileo` problems and two of four `BMC-alpha` problems. All other benchmark problems in the `BMC-IBM` class are easy for both solvers and so are omitted from the table. The other two `BMC-alpha` instances are harder than the two reported even for MiniSat before and after preprocessing. The problems presented in Table 3 are satisfiable.

Each of these bounded model checking problems is brought within the range of Dew_Satz by some form of preprocessing. In general, HyPre and 3-Resolution are the best for this purpose, especially when used together, though on problem `BMC-IBM-13` they are ineffective without the additional use of NiVER. The column showing the number of times Dew_Satz backtracks is worthy of note. In many cases, preprocessing reduces the problem to one that can be solved without backtracking. Solving "without backtracking" has to be interpreted with care here, of course, since a nontrivial amount of lookahead may be required in a "backtrack-free" search. The results for `BMC-galileo-9` furnish a good example of this: HyPre takes 407 seconds to refine the problem, following which Dew_Satz spends 90 seconds on lookahead reasoning while constructing the first (heuristic) branch of its search tree, but then that branch leads directly to a solution. Adding 3-Resolution to the preprocessing step does not change the number of variables, and only slightly reduces the number of clauses, but it roughly halves the time subsequently spent on lookahead.

The instance `BMC-alpha-4408` is hard for Dew_Satz even after preprocessing. While MiniSat with multiple preprocessing solves the problem instance with an order of magnitude faster. We can also observe that HyPre brings more benefit than SatELite,

Table 4 shows results for both solvers on a related problem set consisting of formal verification problems taken from the SAT2005 competition. The `IBM-FV-01` problems are satisfiable except for the problem `IBM-FV-01-k10`; the `IBM-FV-26` problems are unsatisfiable. Most of these satisfiable problems are easy for MiniSat, but the unsatisfiable cases show that the SatELite preprocessor (with which MiniSat was paired in the competition) is by far the least effective of the four we consider for MiniSat on these problems. The preprocessor HyPre proved the unsatisfiability of `IBM-FV-01-k10` in 1.27 seconds. 2-SIMPLIFY was not used to simplify the `IBM-FV-26` problems, because it is limited for input formula with maximum 100,000 variables. Again there are cases in which Dew_Satz is improved from a 15,000 second timeout to a one-branch proof of unsatisfiability. Note that the numbers of clauses in these cases are actually increased by the preprocessor 3-Resolution, confirming that the point of such reasoning is to expose structure rather than to reduce problem size.

5.4 A Highly Symmetrical Problem

FPGA routing problem is a higly symmetrical problem that model the routing of wires in the channels of field-programmable integrated circuits [AMS03]. The problem instances used in the experiment, which were artificially designed by Fadi Aloul, are taken from SAT2002 competition.

Without preprocessing to break symmetries, many of the FPGA routing problems are hard—harder for CDCL solvers than for lookahead-based ones. Not only do they have many symmetries, but the clause graphs are also disconnected. Lookahead techniques with neighbourhood variables ordering heuristic seem able to choose inferences within one graph component before moving to another, whereas MiniSat jumps frequently between components. Table 5 shows performances of both solvers on FPGA routing problem set. Of 21 selected satisfiable (`bart`) problems, MiniSat solves 8 in some 2 hours. It manages better with the unsatisfiable (`homer`) instances, solving 14 of 15 in a total time of around 6 hours. Dew_Satz solves all of the `bart` problems in 17.5 seconds and the `homer` ones in 45 minutes.

The detailed results for two of the satisfiable problems and two unsatisfiable ones (Table 6) are interesting. The resolution-based preprocessors do not give any modification to the size of the input formula except when using SatELite. The Shatter preprocessor, which removes certain symmetries, is tried on its own and in combination with the five resolution-based preprocessors. It should be noted that the addition of symmetry-breaking clauses increases the sizes of the problems, but of course it greatly reduces the search spaces in most cases.

The performance of Dew_Satz after preprocessing is often worse in terms of time than it was before, though there is always a decreases in the size of its search tree. This is because of the increase in the problem size which increases the amount of lookahead process. MiniSat, by contrast, sometimes speeds up by several orders of magnitude after preprocessing.

Instance	Dew_Satz			MiniSat		
	#Solved	Stime	#BackT	#Solved	Stime	#Conflict
bart (21 SAT)	21	17.52	1,536,966	8	7,203	119,782,466
homer (15 UNSAT)	15	2,662	109,771,200	14	22,183	143,719,166

Table 5: Dew_Satz and MiniSat performance, without preprocessing, on FPGA routing problems.

5.5 Order of Preprocessors

Table 7 illustrates the difficulty of selecting the order in which to apply multiple preprocessors. It shows results on just two sample problems. The first is the bounded model checking problem `BMC-IBM-12`, which Dew_Satz attempted with the three preprocessors HyPre, NiVER and 3-Resolution in different orders. Only one order, NiVER followed by HyPre followed by 3-Resolution, renders the problem feasible for Dew_Satz. With the preprocessors in that order, it is solved in less than 2 minutes; with any other order it cannot be solved in more than four hours. The second problem, `ferry10_ks99a`, shows the range of different outcomes produced by varying the order of four preprocessors. If we get it right, we get a solution in 5 seconds, but we know of no simple rule for getting it right in such a case. Neither running NiVER first nor running 3-Resolution last is sufficient. Even with NiVER, HyPre and 3-Resolution in the right order, putting 2-SIMPLIFY first rather than third changes the runtime from 5 seconds to several hours. The third experiment illustrates the efffect of alternating two preprocessors. Simplifying

Instance	Prep.	#Vars/#Cls/#Lits	Ptime	Dew_Satz		MiniSat	
				Stime	#BackT	Stime	#Conflict
bart28	Orig	428/2907/7929	n/a	0.00	0	>15,000	n/a
	Sat	413/2892/11469	0.06	0.02	0	>15,000	n/a
	Sha	1825/8407/27003	0.37	0.06	9	198	775,639
	Sha+3Res	1764/7702/24400	0.46	0.04	1	2,458	7,676,459
	Sha+Hyp	1764/8349/26138	0.41	0.05	20	>15,000	n/a
	Sha+Niv	1781/8358/26759	0.38	0.05	6	5.46	53,683
	Sha+Sat	1728/8254/30422	0.53	0.10	0	115	684,272
	Sha+2Sim	1750/7892/24682	0.39	0.05	17	19.12	150,838
bart30	Orig	485/3617/9954	n/a	0.31	20,160	>15,000	n/a
	Sat	468/3600/14544	0.08	0.03	0	>15,000	n/a
	Sha	2017/9649/30874	0.49	0.11	96	>15,000	n/a
	Sha+3Res	1945/8686/27492	0.60	0.12	224	4,149	7,594,231
	Sha+Hyp	1945/9348/29218	0.54	11,729	28,270,212	>15,000	n/a
	Sha+Niv	1969/9599/30625	0.50	0.06	1	>15,000	n/a
	Sha+Sat	1776/8830/33533	0.77	0.12	1	>15,000	n/a
	Sha+2Sim	1919/8758/27287	0.51	0.05	9	>15,000	n/a
homer19	Orig	330/2340/4950	n/a	473	19,958,400	10,233	51,960,410
	Sat	300/2310/8400	0.04	>15,000	n/a	5,621	54,469,568
	Sha	1460/6764/20242	0.16	2,345	4,828,639	2.26	33,492
	Sha+3Res	1388/5748/16914	0.23	3,231	7,189,966	1.14	21,669
	Sha+Hyp	1387/6547/18865	0.20	4,179	9,611,768	1.90	30,418
	Sha+Niv	1412/6715/19993	0.17	2,570	5,202,084	3.15	45,484
	Sha+Sat	1201/5846/19288	0.34	4,071	6,236,966	1.48	26,517
	Sha+2Sim	1348/5639/16110	0.17	307	678,425	0.70	14,682
homer20	Orig	440/4220/8800	n/a	941	19,958,400	>15,000	n/a
	Sat	400/4180/15200	0.08	1,443	6,982,425	11,448	57,302,582
	Sha	1999/10340/29988	0.28	369	350,610	1.83	22,950
	Sha+3Res	1907/8793/25027	0.37	362	405,059	1.41	18,273
	Sha+Hyp	1905/10527/29129	0.34	1,306	1,451,567	1.10	13,927
	Sha+Niv	1941/10276/29671	0.29	379	349,842	0.91	13,543
	Sha+Sat	1723/9420/30986	0.54	822	300,605	1.00	13,831
	Sha+2Sim	1879/9419/26188	0.31	114	120,297	0.40	6,612

Table 6: Dew_Satz and MiniSat performance, before and after preprocessing, on selected FPGA routing instances.

Instance	Prep.	#Vars/#Cls/#Lits	Ptime	Stime	#BackT
bmc-ibm-12	Hyp+3Res+Niv	10805/83643/204679	96.11	>15,000	n/a
	Niv+Hyp+3Res	12001/100114/253071	85.81	106	6
	3Res+Hyp+Niv	10038/82632/221890	89.56	>15,000	n/a
	3Res+Niv+Hyp	11107/99673/269405	58.38	>15,000	n/a
ferry10_ks99a	2Sim+Niv+Hyp+3Res	1518/32206/65806	0.43	>15,000	n/a
	Niv+3Res+2Sim+Hyp	1532/25229/51873	0.49	11,345	17,778,483
	3Res+2Sim+Niv+Hyp	1793/20597/42365	0.56	907	1,172,964
	Niv+Hyp+2Sim+3Res	1532/24524/50463	0.54	5.19	3,949
ferry10_ks99a	2Sim+Niv	1518/27554/56565	0.08	>15,000	n/a
	2Sim+Niv+2Sim	1518/18988/39433	0.27	3,197	6,066,241
	2Sim+Niv+2Sim+Niv	1486/18956/39429	0.29	129	290,871
	2Sim+Niv+2Sim+Niv+2Sim	1486/23258/48033	0.48	7,355	8,216,100

Table 7: Dew_Satz's performance on instances with preprocessor ordering.

with 2-SIMPLIFY followed by NiVER is insufficient to allow solution before the timeout. Simplifying again with 2-SIMPLIFY brings the runtime down to under an hour; adding NiVER again brings it down again to a couple of minutes; repeating 2-SIMPLIFY, far from improving matters, causes the time to blow out to two hours.

6 Conclusions

We performed an empirical study of the effects of several recently proposed SAT preprocessors on both CDCL and lookahead-based SAT solvers. We describe several outcomes from this study as follow.

1. High-performance SAT solvers, whether they depend on clause learning or on lookahead, benefit greatly from preprocessing. Improvements of four orders of magnitude in runtimes are not uncommon.

2. It is unlikely to equip a SAT solver with just one preprocessor of the kind considered in this paper. Very different preprocessing techniques are appropriate to different problem classes.

3. There are frequently benefits to be gained from running two or more preprocessors in series on the same problem instance.

4. Both clause learning and lookahead need to be enhanced with techniques specific to reasoning with binary clauses, in order to exploit dependency chains, and with techniques for equality reasoning.

5. Lookahead-based solvers also benefit greatly from resolution between longer clauses, as in the 3-Resolution preprocessor. This seems to capture ahead of the search some of the inferences which would be achieved during it by learning clauses. CDCL solvers can also benefit from 3-Resolution preprocessor—dramatically in certain instances—but the effects are far from uniform.

6.1 Future work

The following lines of research are open:

1. It would, of course, be easy if tedious to extend the experiments to more problem sets, more preprocessors and especially to more solvers. We shall probably look at some more DPLL solvers, but do not expect the results to add much more than detail to what is reported in the present paper. One of the more important additions to the class of solvers will be a non-clausal (Boolean circuit) reasoner. We have not yet experimented with such a solver. We have already investigated preprocessing for several state of the art SLS (stochastic local search) solvers, but that is such a different game that we regard it as a different experiment and do not report it here.

2. The more important line of research is to investigate methods for automatically choosing among the available preprocessors for a given problem instance, and

for automaticallly choosing the order in which to apply successive preprocessors. Machine learning may help here, though it would be better, or at least more insightful, to be able to base decisions on a decent theory about the interaction of reasoning methods.

3. Another interesting project is to combine preprocessors not as a series of separate modules but as a single reasoner. For example, it would be possible to saturate under 3-Resolution and hyper-resolution together, in the manner found in resolution-based theorem provers. Whether this would be cost-effective in terms of time, and whether the results would differ in any worthwhile way from those obtained by ordering separate preprocessors, are unknown at this stage.

As SAT solvers are increasingly applied to real-world problems, we expect deductive reasoning by preprocessors to become increasingly important to them.

Acknowledgments

This work was funded by National ICT Australia (NICTA). National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

References

[AMS03] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Design Automation Conference*, pages 836–839. ACM/IEEE, 2003.

[APSS05] Anbulagan, Duc Nghia Pham, John Slaney, and Abdul Sattar. Old resolution meets modern SLS. In *Proceedings of 20th AAAI*, pages 354–359, 2005.

[APSS06] Anbulagan, Duc Nghia Pham, John Slaney, and Abdul Sattar. Boosting SLS performance by incorporating resolution-based preprocessor. In *Proceedings of Third International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS), in conjunction with CP-06*, pages 43–57, 2006.

[AS05] Anbulagan and John Slaney. Lookahead saturation with restriction for SAT. In *Proceedings of 11th CP*, pages 727–731, 2005.

[ASM03] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for boolean satisfiability. In *Proceedings of 18th IJCAI*, Mexico, 2003.

[Bac02] Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of 18th AAAI*, pages 613–619, Edmonton, Canada, August 2002. AAAI Press.

[Bra01] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. In *Proceedings of 17th IJCAI*, pages 515–522, 2001.

[Bra04] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 34(1):52–59, 2004.

[BW04] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Revised Selected Papers of SAT 2003, LNCS 2919 Springer*, pages 341–355, 2004.

[DD01] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of 17th IJCAI*, pages 248–253, Seattle, Washington, USA, 2001.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of 8th SAT, LNCS Springer*, 2005.

[ES03] Niklas Eén and Niklas Sorensson. An extensible SAT-solver. In *Proceedings of 6th SAT*, 2003.

[HvM04] Marijn Heule and Hans van Maaren. Aligning CNF- and equivalence-reasoning. In *Proceedings of 7th SAT*, Vancouver, Canada, 2004.

[HvM06] Marijn Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, (2):47–59, 2006.

[LA97] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of 3rd CP*, pages 341–355, Schloss Hagenberg, Austria, 1997.

[Li00] Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of 17th AAAI*, pages 291–296, USA, 2000. AAAI Press.

[LMS01] I. Lynce and J. Marques-Silva. The interaction between simplification and search in propositional satisfiability. In *Proceedings of CP'01 Workshop on Modeling and Problem Formulation*, 2001.

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC*, pages 530–535, 2001.

[OGMS02] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Recovering and exploiting structural knowledge from CNF formulas. In *Proceedings of 8th CP*, pages 185–199, 2002.

[Qui55] W. V. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62:627–631, 1955.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Rya04] Lawrence O. Ryan. *Efficient Algorithms for Clause Learning SAT Solvers*. PhD thesis, Simon Fraser University, Burnaby, Canada, 2004.

[SE05] Niklas Sorensson and Niklas Eén. MINISAT v1.13 - A SAT solver with conflict-clause minimization. In *2005 International SAT Competition website: http://www.lri.fr/∼simon/contest05/results/descriptions/solvers/minisat_static.pdf*, 2005.

[SKM97] Bart Selman, Henry Kautz, and David McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of 15th IJCAI*, pages 50–54, Nagoya, Aichi, Japan, 1997.

[SP05] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non increasing variable elimination resolution for preprocessing SAT instances. In *Revised Selected Papers of SAT 2004, LNCS 3542 Springer*, pages 276–291, 2005.

[WvM98] Joost P. Warners and Hans van Maaren. A two-phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters*, 23:81–88, 1998.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of International Conference on Computer Aided Design ICCAD2001*, 2001.