# Formula Caching in DPLL



PAUL BEAME
University of Washington
RUSSEll IMPAGLIAZZO
University of California, San Diego
TONIANN PITASSI
University of Toronto
and
NATHAN SEGERLIND
Intel Corporation

We consider extensions of the DPLL approach to satisfiability testing that add a version of *memoization*, in which formulas that the algorithm has previously shown to be unsatisfiable are remembered for later use. Such *formula caching* algorithms have been suggested for satisfiability and stochastic satisfiability by several authors. We formalize these methods by developing extensions of the fruitful connection that has previously been developed between DPLL algorithms for satisfiability and tree-like resolution proofs of unsatisfiability. We analyze a number of variants of these formula caching methods and characterize their strength in terms of proof systems. These proof systems are new and simple, and have a rich structure. We compare them to several studied proof systems: tree-like resolution, regular resolution, general resolution, $\mathsf{Res}(k)$, and Frege systems and present both simulation and separations. One of our most interesting results is the introduction of a natural and implementable form of DPLL with caching, $\mathsf{FC}^{\mathsf{W}}_{reason}$. This system is surprisingly powerful: we prove that it can polynomially simulate regular resolution, and furthermore, it can produce short proofs of some formulas that require exponential-size resolution proofs.

P. Beame's research was supported by NSF grants CCR-0098066 and ITR-0219468. R. Impagliazzo's research was supported by NSF grant CCR-0098197. T. Pitassi's research was supported by NSERC and an Ontario PREA Award. N. Segerlind's research was supported by NSF grants DMS-0100589 and CCR-0098197.

Authors' addresses: P. Beame, Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350; email: beame@cs.washington.edu; R. Impagliazzo, Computer Science and Engineering, University of California, San Diego, CA 92093-0404; email: russell@cs.ucsd.edu; T. Pitassi, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 3G4; email: toni@cs.toronto.edu; N. Segerlind, Intel Corporation, Hillsboro, OR 97124; email: nathan.l.segerlind@intel.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1942-3454/2010/03-ART9 $10.00 DOI: 10.1145/1714450.1714452.
http://doi.acm.org/10.1145/1714450.1714452.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Complexity of proof procedures*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*Backtracking*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Resolution*

General Terms: Theory, Algorithms

Additional Key Words and Phrases: Resolution, satisfiability, proof complexity

**ACM Reference Format:**
Beame, P., Impagliazzo, R., Pitassi, T., and Segerlind, N. 2010. Formula caching in DPLL. ACM Trans. Comput. Theor. 1, 3, Article 9 (March 2010), 33 pages. DOI = 10.1145/1714450.1714452. http://doi.acm.org/10.1145/1714450.1714452.

## 1. INTRODUCTION

Over the last decade, many variations and extensions of DPLL have been introduced (both for satisfiability and stochastic satisfiability). A generally useful idea is to store intermediate results for later reuse as the DPLL tree is searched. The technique of *clause learning*, for which there have been many good implementations [Marques-Silva and Sakallah 1996; Zhang 1997; Moskewicz et al. 2001; Zhang et al. 2001] and which has revolutionized practical satisfiability solving, can be viewed as a form of *memoization* (saving solved subproblems, also called *caching*) of DPLL. In clause learning, the algorithm stores, in the form of learned clauses, partial assignments that force contradictions and uses these learned clauses to augment the clauses of the original formula. This technique, which can be efficiently simulated by resolution, has been studied from the point of view of proof complexity by Beame et al. [2004]. More generally, memoization is useful in a variety of backtracking algorithms. As one example, Robson [1986] uses memoization to speed up a backtracking algorithm for maximum independent set.

The methods that interest us here involve caching *unsatisfiable residual formulas* rather than partial assignments. They were first defined by Majercik and Littman [1998] where DPLL-based algorithms with caching are studied and implemented to solve large probabilistic planning problems. In that paper, there were no analytic runtime guarantees, although the empirical results were very promising. More recently, Bacchus et al. [2003b; 2003a] defined DPLL-based algorithms with caching for counting satisfying assignments and Bayesian inference and gave time and space bounds that are as good as any known algorithm for these problems in terms of a connectivity measure of the underlying set of clauses/Bayes network.

Thus, while applications of memoization in many different guises for DPLL have been studied in the past, this article is the first to specifically formalize proof systems for SAT based on adding memoization of residual formulas to DPLL, and to analyze the complexity of these systems. We present several different ways to introduce caching of unsatisfiable residual formulas into DPLL algorithms. We characterize the strength of these nondeterministic algorithms in terms of proof systems. Then we compare these proof systems to each other and to standard proof systems. This gives a fairly complete picture of the relative strengths of the various approaches.

```
DPLL(F){
    If F is empty
        Report satisfiable and halt
    If F is contains the empty clause Λ
        return
    Else choose a literal x
        DPLL(F|_x)
        DPLL(F|_x̄)}
```

Fig. 1. Generic DPLL algorithm.

Many of our results are surprising, since at first glance it seems that adding memoization to DPLL cannot strengthen the system beyond resolution. One of our most interesting results is the introduction of a natural and implementable form of DPLL with caching, $\mathsf{FC}^{\mathsf{W}}_{reason}$. This system is surprisingly powerful; we prove that it can produce short proofs of some formulas that require exponential-size resolution proofs. Thus, adding formula caching to DPLL is potentially much more powerful than clause learning, as clause learning is a form of resolution.

As mentioned before, our results characterize the relative strengths of various extensions of DPLL in terms of proof systems. Here we continue the fruitful connection between algorithm design and proof complexity found in the formalization of the DPLL approach to satisfiability testing in terms of tree-like resolution proofs. In doing so, we view DPLL as a meta-algorithm, as shown in Figure 1, whose input is a CNF formula $F$. As written, the step "*choose* a literal $x$" is not fully specified. This is one of many examples in algorithm design in which there is a single framework or meta-algorithm with a variety of options for how this meta-algorithm should proceed at a given point in its execution. We can thus think of this meta-algorithm as a *nondeterministic algorithm*, in which the algorithm expresses as a nondeterministic choice from among the options. In devising a deterministic algorithm within this framework, the algorithm designer replaces the nondeterministic choices with deterministic rules.

The nondeterminism only occurs in the step in which the algorithm chooses the branching literal $x$. To create a deterministic DPLL algorithm, a deterministic rule must be given for this choice. In the case of DPLL, such a rule would likely include priority for literals in unit clauses which is equivalent to including explicit unit clause propagation. Beyond this simple preference, many such deterministic rules have been suggested over the years, and the performance of DPLL, empirically, has been found to be quite sensitive to the choice of this rule.

Since there are unlimited numbers of deterministic versions, it seems impossible to exactly analyze all possible variants. However, the performance of the *nondeterministic* version of this algorithm has been characterized in terms of *tree-like resolution*. Tree-like resolution is an example of an abstract *propositional proof system*, which is an efficient method for verifying proofs in propositional logic represented in a given format. A propositional proof system can

be viewed alternatively as a nondeterministic algorithm for accepting propositional tautologies (or, equivalently, refuting contradictions). Proof complexity studies how the lengths (sizes) required for such proofs depend on the proof systems being employed. Lower bounds for the complexity of tree-like resolution refutations [Haken 1985; Chvátal and Szemerédi 1988; Ben-Sasson and Wigderson 2001] then can be used to prove the limitations of any deterministic instantiation of DPLL. Although proof complexity studies refutation and proof, as shown by Achlioptas et al. [2004], the time required by backtracking satisfiability algorithms like DPLL is directly dependent on their efficiency as proof systems. Thus, our surprising results on the efficiency of the formula caching variants viewed as proof systems suggest that formula caching holds promise for satisfiability testing as well.

### 1.1 Outline of Results

We describe our results in terms of the known hierarchy of resolution-like proof systems: DPLL, which is equivalent to tree-like resolution; regular resolution (REG); general resolution; $\mathsf{Res}(k)$ for each $k \geq 2$; depth-2 Frege ($\mathcal{F}_2$); and extended resolution, which is equivalent to extended Frege ($e\mathcal{F}$). Most of these proof systems are axiomatic, whereby a proof is a sequence of lines, and each line follows from one or two previous lines by one of a fixed set of axiom schemas. But as is the case with DPLL, nondeterministic satisfiability algorithms can also be viewed as proof systems so we often refer to such algorithms as proof systems. Proof systems can be related by a notion of efficient simulation, called p-simulation, which says that efficient proofs in one system can be translated to efficient proofs in another system. The definitions of proof complexity, these proof systems, and p-simulation are given in Section 3.1. The preceding hierarchy is known to be strict under p-simulation; in fact, exponential gaps in efficiency are known between each of its levels.

In Section 2, we describe the various variants of memoized DPLL algorithms that constitute formula caching. We begin with a basic extension of DPLL to include a cache of known unsatisfiable formulas, called $\mathsf{FC}$. This checks its input for membership in the cache before proceeding with the recursive call. We also define extensions of this system that include more complicated, but theoretically still efficiently implementable, checks than membership to derive algorithms $\mathsf{FC}^{\mathsf{W}}$ and $\mathsf{FC}^{\mathsf{WS}}$ where $\mathsf{W}$ and $\mathsf{S}$ stand for Weakening and Subsumption, respectively.

In each of these formula caching algorithms, no information other than the cache contents and an indication of failure is available as the result of a recursive call. We also consider an extension of these ideas that allows the recursive call to return more pointed information about the reason for failure. When the resulting algorithm incorporates Weakening and Subsumption, we obtain an algorithm $\mathsf{FC}^{\mathsf{W}}_{reason}$ that is quite natural and is as efficiently implementable as $\mathsf{FC}^{\mathsf{W}}$ but which we show to be much more powerful than $\mathsf{FC}^{\mathsf{W}}$ (or $\mathsf{FC}^{\mathsf{WS}}$).

The formula caching systems with reasons suggest that there is considerable scope for a clever algorithm designer to incorporate memoization in ways that cannot be efficiently simulated in the aforesaid systems. We design

| Proof System | Systems it p-simulates | Systems it cannot p-simulate | Systems that cannot p-simulate it |
|---|---|---|---|
| FC | DPLL | REG(Theorem 4.28) | DPLL (Corollary 4.20) |
| $\mathsf{FC}^{\mathsf{WS}}$ | DPLL | REG(Theorem 4.28) | DPLL (Corollary 4.20) |
| $\mathsf{FC}^{\mathsf{W}}_{reason}$ | REG(Theorem 4.9) | - | Res($k$) (Theorem 4.23) |
| $\mathsf{FC}^{\mathsf{W}}_{nondet}$ | REG(Theorem 4.6) | - | Res($k$) (Theorems 4.23, 4.3) |
| $\mathsf{FC}^{\mathsf{WS}}_{nondet}$ | REG(Theorem 4.6) | - | Res($k$) (Theorems 4.23, 4.3) |

Fig. 2. Relationship of various formula caching proof systems to other resolution-like proof systems.

systems to represent the "ultimate limits" of these forms of memoization, $\mathsf{FC}^{\mathsf{W}}_{nondet}$ and $\mathsf{FC}^{\mathsf{WS}}_{nondet}$, in which the algorithm nondeterministically anticipates the cached contradictions that will be weakened and/or subsumed to determine unsatisfiability of its input. It would be highly nontrivial to incorporate these features into an existing DPLL algorithm. However, we feel that any algorithm that somehow incorporated memoization of cached contradictions into DPLL would probably be efficiently simulated by our $\mathsf{FC}_{nondet}$ systems. Thus, bounds on the strength of the $\mathsf{FC}_{nondet}$ systems are bounds on the potential of the memoization technique.

From their definitions it is clear that as proof systems with the same options T for Weakening and Subsumption, $\mathsf{FC}^{\mathsf{T}}$ is p-simulated by $\mathsf{FC}^{\mathsf{T}}_{reason}$ which is in turn p-simulated by $\mathsf{FC}^{\mathsf{T}}_{nondet}$ and if neither Weakening nor Subsumption is allowed then the $\mathsf{FC}_{reason}$ and $\mathsf{FC}_{nondet}$ coincide with FC.

In Section 3, after giving a detailed overview of proof complexity definitions and the standard axiomatic proof systems related to resolution we define our new *contradiction caching* axiomatic proof systems CC+T. We then relate these contradiction caching proof systems to the FC algorithms viewed as proof systems and show that they are equivalent to the corresponding $\mathsf{FC}^{\mathsf{T}}_{nondet}$ proof systems. In Section 4, we compare these systems to each other and to the standard resolution-like proof systems. For the most interesting systems, our results can be summarized in Figure 2. In Section 5, we study a generalization of the CC and FC systems where we add a simple form R (for Restriction) of the substitution rule, and prove that with this addition, it is p-equivalent to extended Frege. We conclude in Section 6 with related results and future directions.

### History and Errata

The present article is based on a conference paper [Beame et al. 2003] in which we more prominently used the Restriction rule (R), discussed here in Section 5, in order to efficiently simulate general resolution. In that article we incorrectly claimed that the contradiction caching proof systems CC and the formula

caching proof system $\mathsf{FC}^{\mathsf{WS}}_{reason}$ involving both Subsumption and Restriction have the subformula property. It turns out that of these only CC and CC+W have the subformula property. The subformula property, which is also shared by FC, $\mathsf{FC}^{\mathsf{W}}$, and our revised $\mathsf{FC}^{\mathsf{W}}_{reason}$, is critical for our present Lemma 4.15 and the resulting exponential separations between proof systems.

The fact that Restriction does not have the subformula property makes it much more powerful than we had anticipated. It was brought to our attention that using contradiction caching with this rule one can derive a renaming rule and thus, for example, obtain efficient proofs of the pigeonhole principle, contrary to our mistaken claim that all of the CC systems, including the strongest one, CC+WSR, could be p-simulated by depth-2 Frege systems. The "R" rule is in fact quite powerful. We correct our error by proving in Section 5 that CC+WSR is actually p-equivalent to the Extended Frege proof system.

In addition to the aforesaid changes, we have modified the definitions of the formula caching proof systems that extend FC so that the algorithms are not required to check the cache (or add a formula to the cache) on each recursive call. This better reflects what one would do in practice but it also seems essential for the system $\mathsf{FC}^{\mathsf{W}}_{reason}$ to simulate regular resolution, as we now show in Lemma 4.9.

## 2. MEMOIZATION AND DPLL: FORMULA CACHING

Memoization means saving previously solved subproblems and using them to prune a backtracking search. In the satisfiability algorithms we consider, this will mean storing a list of previously refuted formulas and checking whether the unsatisfiability of some formula in the list allows us to conclude easily, before branching, that our current formula is unsatisfiable.

A pure backtracking algorithm usually corresponds to a tree-like proof system, since the recursive refutations are done independently and not reused. Our original intuition was that introducing memoization into a backtracking algorithm would move from a tree-like proof system to the corresponding DAG-like system. However, the real situation turns out to be somewhat more complicated. There are actually several reasonable ways to introduce memoization into DPLL. None of them seem to be equivalent to DAG-like resolution, and many move beyond resolution.

—*Basic Formula Caching.* The basic idea of the simplest memoized version of the DPLL algorithm is, as mentioned before, to record the unsatisfiable residual formulas found over the course of the algorithm in a list and before applying recursion to include checking the list to see if $F$ is already known to be unsatisfiable. This yields the algorithm of Figure 3 where $L$ is the cache of residual formulas known to be unsatisfiable. Satisfiability is determined by calling $\mathsf{FC}(F,\emptyset)$.

While we present FC as a nondeterministic algorithm, one can also view it as a simple transformation for deterministic DPLL algorithms. We simply replace the nondeterministic branching rule with the rule used by the DPLL algorithm and add some heuristics for *Cache-Add* and *Cache-Check* that would decide for the purposes of memory and time efficiency whether or not to cache

```
FC(F, L){
  If F is empty
      Report satisfiable and halt
  If F contains the empty clause Λ
      return
  // Optionally check if L trivially implies that F is unsatisfiable
  Else If Cache-Check and F is in L
      return
  Else choose a literal x
      FC(F|x,L)
      FC(F|x̄,L)
      If Cache-Add
        Add F to L}
```

Fig. 3. The basic Formula Caching algorithm. *Cache-Check* and *Cache-Add* determine whether or not to check the cache for $F$ or add $F$ to the cache, respectively.

a restricted formula and would determine whether it is worthwhile checking the cache. Checking the cache would be particularly simple using some form of hash table.

This is a straightforward way of adding memoization to DPLL, similar to other uses of memoization in backtracking for other problems. For example, Robson's maximum independent set algorithm maintains a cache of medium-size subgraphs with known bounds on their maximum independent set sizes, and checks if the current subgraph is in the cache.

We call the preceding nondeterministic algorithm, viewed as a proof system, FC. It is obviously at least as powerful as DPLL, since the presence of the cache only prunes branches, never creates them.

—*Formula Caching with Weakening.* Once we have the notion that we are checking the formula $F$ against a cache of known unsatisfiable formulas there are other natural related checks that we might do. For example, it may be the case that $F$ contains all the clauses of some formula in the list $L$. We can check this in time that is polynomial as a function of the size of the formula $F$ and list $L$. We call such a test a *Weakening* test. This leads to the algorithm $\mathsf{FC}^{\mathsf{W}}$ given in Figure 4.

—*Formula Caching with Weakening and Subsumption.* There is another way that the unsatisfiability of $F$ can trivially follow from that of some formula in $L$. Given clauses $C$ and $D$ such that $C$ *subsumes* $D$ (i.e., $C \subset D$) we have that $C$ is a stronger constraint than $D$. Therefore adding a subsumption test to Weakening we obtain the $\mathsf{FC}^{\mathsf{WS}}$ algorithm given in Figure 5 where the check whether $L$ trivially implies $F$ asks *"Is there is a formula $G$ in $L$ such that every clause of $G$ contains a clause of $F$?"* Again this is polynomial as a function of the sizes of $F$ and $L$.

Weakening and Subsumption are very natural additions to a memoized backtracking algorithm. Among other benefits, they allow a limited amount of "without loss of generality" reasoning in addition to logical implications of the constraints, because branches dominated by earlier ones get pruned. To see how they can capture such "without loss of generality" reasoning, it is

```
FC^W(F, L){
    If F is empty
        Report satisfiable and halt
    If F contains the empty clause Λ
        return
    // Optionally check if L trivially implies that F is unsatisfiable
    Else If Cache-Check and F contains all clauses of some formula in L
        return}
    Else choose a literal x
        FC^W(F|_x, L)
        FC^W(F|_x̄, L)
        If Cache-Add
          Add F to L}
```

Fig. 4. Formula Caching with Weakening.

```
FC^WS(F, L){
    If F is empty
        Report satisfiable and halt
    If F contains the empty clause Λ
        return
    // Optionally check if L trivially implies that F is unsatisfiable
    Else If Cache-Check and there is a G ∈ L such that every clause of G contains a clause of F
        return
    Else choose a literal x
        FC^WS(F|_x, L)
        FC^WS(F|_x̄, L)
        If Cache-Add
          Add F to L}
```

Fig. 5. Formula Caching with Weakening and Subsumption.

convenient to consider another context. For example, consider a simple backtracking algorithm for finding an independent set $S$ of size $k$ in a graph $G$, branching on a node $x$ with one neighbor $N(x) = \{y\}$. We will argue informally, using a memoized backtracking algorithm with Weakening and Subsumption, that without loss of generality, the algorithm should include $x$ in the set. The algorithm first branches on whether $x \in S$, then on whether $y \in S$, exploring the $x \in S$ branch first. The branch $x \in S$ forces $y \notin S$, so the subproblem is to find an independent set of size $k-1$ in $G - \{x, y\}$. Assume that this recursive search fails. The branch $x \notin S$, $y \notin S$ is to find an independent set of size $k$ in $G - \{x, y\}$, a strengthening of the failed branch that gets pruned by Weakening and Subsumption. The final branch $x \notin S, y \in S$ is to find an independent set of size $k-1$ in $G - \{x, y\} - N(y)$, again a strengthening of the failed branch. Only the branch where $x \in S$ gets recursively explored.

As the previous example illustrates, when we have weakening and subsumption, the order in which the algorithm explores branches matters. So, in addition to a deterministic branching rule, we would need a heuristic to determine the order of branches to construct a deterministic version of $\mathsf{FC}^{\mathsf{WS}}$.

*—Formula Caching with Returned Reasons for Unsatisfiability.* One drawback of even the strongest basic system, $\mathsf{FC}^{\mathsf{WS}}$ is that some potentially useful

$\mathsf{FC}^{\mathsf{W}}_{reason}(F, L)\{$
**If** $F$ is empty
Report satisfiable and **halt**
**If** $F$ contains the empty clause $\Lambda$
**return**$(\Lambda)$
*// Optionally check if $L$ trivially implies that $F$ is unsatisfiable*
**Else If** *Cache-Check* and $F$ contains every clause of some formula in $L$
*Select* some such formula $J$ in $L$
**return**$(J)$
**Else** *choose* a literal $x$
$G \leftarrow \mathsf{FC}^{\mathsf{W}}_{reason}(F|_x, L)$
$H \leftarrow \mathsf{FC}^{\mathsf{W}}_{reason}(F|_{\overline{x}}, L)$
$I \leftarrow \bigwedge_{C \in F \cap G \cap H} C$
$J \leftarrow I \wedge \bigwedge_{C \in (G \cap H) \setminus I} (x \vee C) \wedge (\overline{x} \vee C) \wedge \bigwedge_{C \in G \setminus H} (\overline{x} \vee C) \wedge \bigwedge_{C \in H \setminus G} (x \vee C)$
**If** *Cache-Add*
Add $J$ to $L$
**return**$(J)\}$

Fig. 6. Formula Caching with returned reasons for unsatisfiability.

information about unsatisfiable formulas may be available to be learned but may be lost on the return from a recursive call. For example, if for some formula $F$ the restricted formula $F|_x$ has a small unsatisfiable subformula $G$ and $F|_{\overline{x}}$ has a small unsatisfiable subformula $H$ then $F$ will have a small subformula whose restrictions under $x$ and $\overline{x}$ contain $G$ and $H$, respectively. However, $\mathsf{FC}^{\mathsf{WS}}$ will learn the formula containing all of $F$, not just this subformula. In order to take advantage of this kind of information we can augment the algorithm with a return value consisting of a formula giving a "reason" that $F$ is unsatisfiable. In order to understand the algorithm, consider the following example. Let $F = F_1 \wedge F_2 \wedge F_3$, where $F_1$ are those clauses that contain $\overline{x}$, $F_2$ are those clauses containing $x$, and $F_3$ are clauses that do not contain the variable $x$. Then $F|_x = F'_1 \wedge F_3$ and $F|_{\overline{x}} = F'_2 \wedge F_3$. Suppose we know that $F|_x$ is unsatisfiable because of $G$, where $G = (F'_1)^1 \wedge F^1_3$, $(F'_1)^1 \subseteq F'_1$ and $F^1_3 \subseteq F_3$. Similarly suppose that we know that $F|_{\overline{x}}$ is unsatisfiable because of $H$, where $H = (F'_2)^2 \wedge F^2_3$, $(F'_2)^2 \subseteq F'_2$ and $F^2_3 \subset F_3$. Now since $G$ and $H$ are both unsatisfiable, so is $J = I \wedge A \wedge B$, where $I = F^1_3 \cap F^2_3$, $A$ is $(F'_1)^1$ with $\overline{x}$ added back, and $B$ is $(F'_2)^2$ with $x$ added back. Thus $J$ is added to the cache as the "reason" for the unsatisfiability of $F$. We describe the algorithm as an extension of $\mathsf{FC}^{\mathsf{W}}$ in Figure 6. We will see that this is strong enough to simulate regular resolution efficiently.

—*Formula Caching with Nondeterministic Rules.* Given that we are using a cache of unsatisfiable formulas to prove that a formula is unsatisfiable, we may wish to apply the rules such as weakening, or subsumption a little earlier in the process so that we can be more efficient at generating formulas that we previously have seen to be unsatisfiable. We could, for example, allow the algorithm to nondeterministically apply weakening at any point in the algorithm. This is a generalization of the usual pure literal rule of DPLL which allows one to remove clauses containing a literal that occurs only positively (or only negatively) in the formula. (Of course, a bad early choice of weakening may suggest

```
FC^WS_nondet(F, L){
    If F is empty
        Report possibly satisfiable and halt
    //Non-deterministic reverse weakening
    Remove some subset of clauses of F (possibly none)
    //Non-deterministic reverse subsumption
    For each clause of F, add some variables (possibly none)
    // Check if L trivially implies that F is unsatisfiable
    If F contains the empty clause Λ or F is in L
        return
    Else choose a literal x
        FC^WS_nondet(F|_x, L)
        FC^WS_nondet(F|_x̄, L)
        Add F to L}
```

Fig. 7. Formula Caching with nondeterministic application of Weakening and Subsumption.

satisfiability when that is not the case, but the system will remain sound for proofs of unsatisfiability.) Similarly, we can define an algorithm $\mathsf{FC}^{\mathsf{WS}}_{nondet}$ that, as well as allowing the removal of clauses, also allows any clause of $F$ to be weakened by adding extra literals to it. We give a description of $\mathsf{FC}^{\mathsf{WS}}_{nondet}$ in Figure 7; the other algorithms can be obtained by deleting appropriate lines.

If $\mathsf{FC}^{\mathsf{WS}}_{nondet}$ completes without reporting that $F$ is possibly satisfiable then $F$ will be unsatisfiable. It is immediate that as a refutation system $\mathsf{FC}^{\mathsf{WS}}_{nondet}$ is at least as powerful as $\mathsf{FC}^{\mathsf{WS}}$. It could possibly be more powerful, since the weakened formula is remembered for later use. Similarly, $\mathsf{FC}^{\mathsf{W}}_{nondet}$ efficiently simulates $\mathsf{FC}^{\mathsf{W}}$.

It may seem that some of these new systems allowing nondeterministic manipulation of $F$ itself are a little unnatural. However, we shall see that they correspond directly to the extremely natural contradiction caching inference systems for unsatisfiable CNF formulas that we define in the next section. Also, reasoning about such systems covers many algorithms that prune searches based on reasoning that identifies unnecessary constraints, such as the pure literal rule or its generalization to autarchs [Monien and Speckenmeyer 1985], or deleting a node of degree 2 or less from a 3-coloring problem. While such weakening only guides the choice of branching variables in a pure backtracking search, caching the simplified formula may make a more dramatic difference. In fact, we shall see that $\mathsf{FC}^{\mathsf{W}}_{nondet}$ is surprisingly powerful; in particular it is capable of refuting formulas that are hard for systems more powerful than resolution.

## 3. AXIOMATIC PROOF SYSTEMS

### 3.1 Proof Complexity

We review the basic definitions of proof complexity and give some important examples of propositional proof systems. Propositional proof complexity is often defined in terms of proofs of tautologies but, since $\varphi$ is a tautology if and only

if $\neg\varphi$ is unsatisfiable, propositional proof systems are equivalently stated in terms of proofs of unsatisfiability (refutations) of propositional formulas. Furthermore, following the usual arguments that it suffices to decide satisfiability for CNF formulas, we obtain the following standard definition.

*Definition* 3.1. A *propositional proof system* for refuting CNF formulas is a polynomial-time algorithm V (a verifier) such that for all CNF formulas $\varphi$, $\varphi$ is unsatisfiable if and only if there exists a string $\Pi$ (a V-refutation of $\varphi$) such that V accepts input $(\varphi, \Pi)$.

This definition is very similar to the standard definition of verifiers for NP except that it allows the algorithm's running time to be polynomial in the size of $\Pi$ and does not place any limit on this size. We often specify a proof system V simply by describing a format for its V-refutations, assuming that this format is easy to check.

The following definition allows us to define and compare the efficiency or power of proof systems.

*Definition* 3.2. Given a proof system V for refuting unsatisfiable CNF formulas, let $s_{\mathsf{V}}(F)$ be the minimum size (length) of a V-refutation of $F$. For two refutation/proof systems $\mathsf{V}_1$ and $\mathsf{V}_2$, we say that $\mathsf{V}_2$ *p-simulates* $\mathsf{V}_1$ if for every unsatisfiable formula $F$, if there is a $\mathsf{V}_1$ refutation of $F$ of length $s$, then there is also a $\mathsf{V}_2$ refutation of $F$ of size polynomial in $s$ and the size of $F$. $\mathsf{V}_1$ and $\mathsf{V}_2$ are *p-equivalent* if $\mathsf{V}_1$ p-simulates $\mathsf{V}_2$ and conversely, $\mathsf{V}_2$ p-simulates $\mathsf{V}_1$.

Any complete (deterministic or nondeterministic) algorithm $A$ for SAT corresponds to a propositional proof system $\mathsf{V}_A$ whose refutations have size essentially equal to the running time of $A$ on unsatisfiable formulas: The $\mathsf{V}_A$-refutation of such a formula $F$ is a transcript of the execution of $A$ that fails to find an assignment for $F$. (The transcript size is actually the product of the time and the space used by the algorithm.) $\mathsf{V}_A$ simply checks that this transcript correctly follows $A$. For simplicity we use $A$ itself to refer to this proof system. One such example is DPLL; a *DPLL refutation* of an unsatisfiable $F$ is a complete DPLL search tree on input $F$.

A *resolution* refutation of a CNF formula $F$ is a sequence of clauses $C_1, \ldots, C_r = \Lambda$ where $\Lambda$ is the empty clause and each $C_i$ is either a clause of $F$ or follows from two previous clauses $C_j, C_k$ for $j, k < i$ by the *resolution rule*, which says that for any variable $x$ and any disjunctions of literals $A$ and $B$ one can derive the clause $(A \vee B)$. from clauses $(A \vee x)$ and $(B \vee \neg x)$. (This derivation is called "resolving on $x$" and $(A \vee B)$ is called the resolvent.) Since resolution is sound and complete it forms a propositional proof system for refuting CNF formulas.

The inferences in the resolution refutation form a directed acyclic graph (DAG): the nodes are the $C_1, \ldots, C_r = \Lambda$ and for each $C_i$ derived from $C_j$ and $C_k$ there are edges from $C_i$ to $C_j$ and $C_k$. A *tree-like resolution* refutation is a resolution refutation in which this graph forms a directed tree (each node has indegree at most one). It is well known that as a proof system DPLL is equivalent to tree-like resolution; for example, see Urquhart [1995].

Another natural special case of resolution is called *regular resolution*. In a regular resolution refutation, if a clause $C_i$ is the result of resolving away a variable $x$ then no clause derived from $C_i$ can contain the variable $x$, that is, there is no path in the graph of inferences between two clauses that are the result of resolving on the same variable. Optimal tree-like resolution refutations are regular. Regular resolution is therefore at least as efficient as DPLL but it also covers the original Davis-Putnam proof system [Davis and Putnam 1960] and can be exponentially more efficient than DPLL [Bonet et al. 2000]. In turn, general resolution can be exponentially more efficient than regular resolution [Alekhnovich et al. 2001].

One can define more general inference systems for refuting CNF formulas by allowing inference on more complex objects than clauses. In particular, for positive integer $k$, $\mathsf{Res}(k)$ is a proof system similar to resolution but it allows $k$-DNF formulas as objects instead of clauses. In this system there is an inference rule deriving $(A \vee B_1 \vee \ldots \vee B_k)$ from $(A \vee (x_1 \wedge \ldots \wedge x_k))$ and $(B_1 \vee \neg x_1), \ldots, (B_k \vee \neg x_k)$ and rules for the distributive laws. $\mathsf{Res}(1)$ is easily seen to be the same as resolution. It is known that for any $k$, $\mathsf{Res}(k+1)$ can be exponentially more efficient than $\mathsf{Res}(k)$ [Segerlind et al. 2002].

More general still is the following system $\mathcal{F}_2$ which is a standard depth-2 refutation system (sometimes called depth-2 Frege) for CNF formulas defined by Pitassi and Urquhart [1995]. Note that, unlike resolution, which has an implied conjunction between its clauses, in $\mathcal{F}_2$, each formula in the proof is self-contained and is itself a CNF formula.

*Definition* 3.3 *[Pitassi and Urquhart 1995].* $\mathcal{F}_2$ is a refutation system for CNF formulas. Let $x$ denote a variable; let $A$ and $B$ denote a disjunction of literals, and let $F$ and $G$ denote CNF formulas. $\mathcal{F}_2$ has a single axiom schema, $(x \wedge \overline{x})$, and the following 5 rules:

R0': $A \wedge A \wedge F \rightarrow A \wedge F$
R1': $F \rightarrow F \wedge B$
R2': $(A \vee B) \wedge F \rightarrow A \wedge F$
R3': $(A \wedge F), (B \wedge F) \rightarrow (A \vee B) \wedge F$
R4': $F \wedge (x), G \wedge (\overline{x}) \rightarrow F \wedge G$

An $\mathcal{F}_2$ refutation of a CNF formula $F$ is sequence of CNF formulas, $F_1, F_2, \ldots, F_r = F$, each of which is either an axiom or follows from previous formulas by one of the $\mathcal{F}_2$ inference rules.

Cook and Reckhow [1977], who originally formalized proof complexity, defined two of the most important and general classes of proof systems, Frege and Extended Frege proofs. *Frege* proofs follow the pattern of standard axiomatic inference systems as in $\mathcal{F}_2$ given earlier. However, they allow arbitrary propositional logic formulas rather than being restricted to depth-2 formulas. (As shown by Cook and Reckhow [1977], any sound and implicationally complete sets of inference rules yield equivalent proof systems.) This ability to allow more complicated intermediate formulas yields a proof system that is exponentially more powerful than $\mathcal{F}_2$ [Buss 1987; Beame et al. 1992].

Finally, in addition to the proof rules of Frege systems, *Extended Frege* proofs allow the introduction of new *extension* variables to stand for entire formulas as the proof proceeds. This introduction is standard in mathematical arguments. Extension variables may make a proof much more concise and can be viewed as allowing Boolean circuits as objects in proofs. Simply allowing these extension variables is very powerful: By augmenting ordinary resolution with an extension rule, one derives a proof system called Extended Resolution which is equivalent in power to Extended Frege proof systems.

### 3.2 Contradiction Caching Inference Systems

We now define several inference systems for unsatisfiable formulas that are closely related to some of the formula caching algorithms in the previous section. The objects of these proof systems will be Conjunctive Normal Form (CNF) formulas. CNF formulas will be assumed to be sets of clauses and clauses will be assumed to be sets of literals so the order of clauses and of literals within each clause is immaterial. In the following, $x, y, z$ denote literals which can be variables or their negations, $\varphi$, $\psi$ will denote CNF formulas, and $C, D, E$ will denote clauses. (A clause also can be viewed as simple case of a CNF formula.) The (unsatisfiable) empty clause will be denoted $\Lambda$.

*Definition* 3.4. Given a CNF formula $\varphi$ and literal $x$ (or $\overline{x}$), the formula $\varphi|_x$ (respectively $\varphi|_{\overline{x}}$) denotes the simplified CNF formula in which all clauses containing $x$ (respectively $\overline{x}$) have been removed and all clauses containing $\overline{x}$ (respectively $x$) are shortened by eliminating that literal. More generally, given a sequence of literals $xyz$, for example, we write $\varphi|_{xyz} = \varphi|_x|_y|_z$ and for a clause $C$ we identify $\overline{C}$ with the sequence of negations of the literals in $C$ and define $\varphi|_{\overline{C}}$ to be the restriction of $\varphi$ in which every literal of $C$ has been set to false.

We define several related proof systems for showing that CNF formulas are unsatisfiable based on the following inference rules.

1. *Axiom.* $\dashv \Lambda$
2. *Branching.* $\varphi|_x, \varphi|_{\overline{x}} \dashv \varphi$ where $x$ is any variable and $\varphi$ is any CNF formula.

3a. *Limited Weakening.* $\Lambda \dashv \Lambda \wedge \psi$ where $\psi$ is any CNF formula.

3. *Weakening.* $\varphi \dashv \varphi \wedge \psi$ where $\varphi$ and $\psi$ are any CNF formulas.
4. *Subsumption.* $\varphi \wedge C \dashv \varphi \wedge D$ where $D \subseteq C$ are clauses and $\varphi$ is any CNF formula.

*Definition* 3.5. A CC *(contradiction caching)* refutation of a CNF formula $F$ is a sequence $\varphi_1, \ldots, \varphi_s = F$ of CNF formulas such that each $\varphi_i$ for $i > 1$ follows from $\varphi_j$, $j < i$ using one of the proof rules (1)–(3a): axiom, Branching, and Weakening. If in addition we allow some forms of the weakening rule (3), or the Subsumption proof rule (4), we denote the system by CC+, together with the appropriate subset of letters W and S.

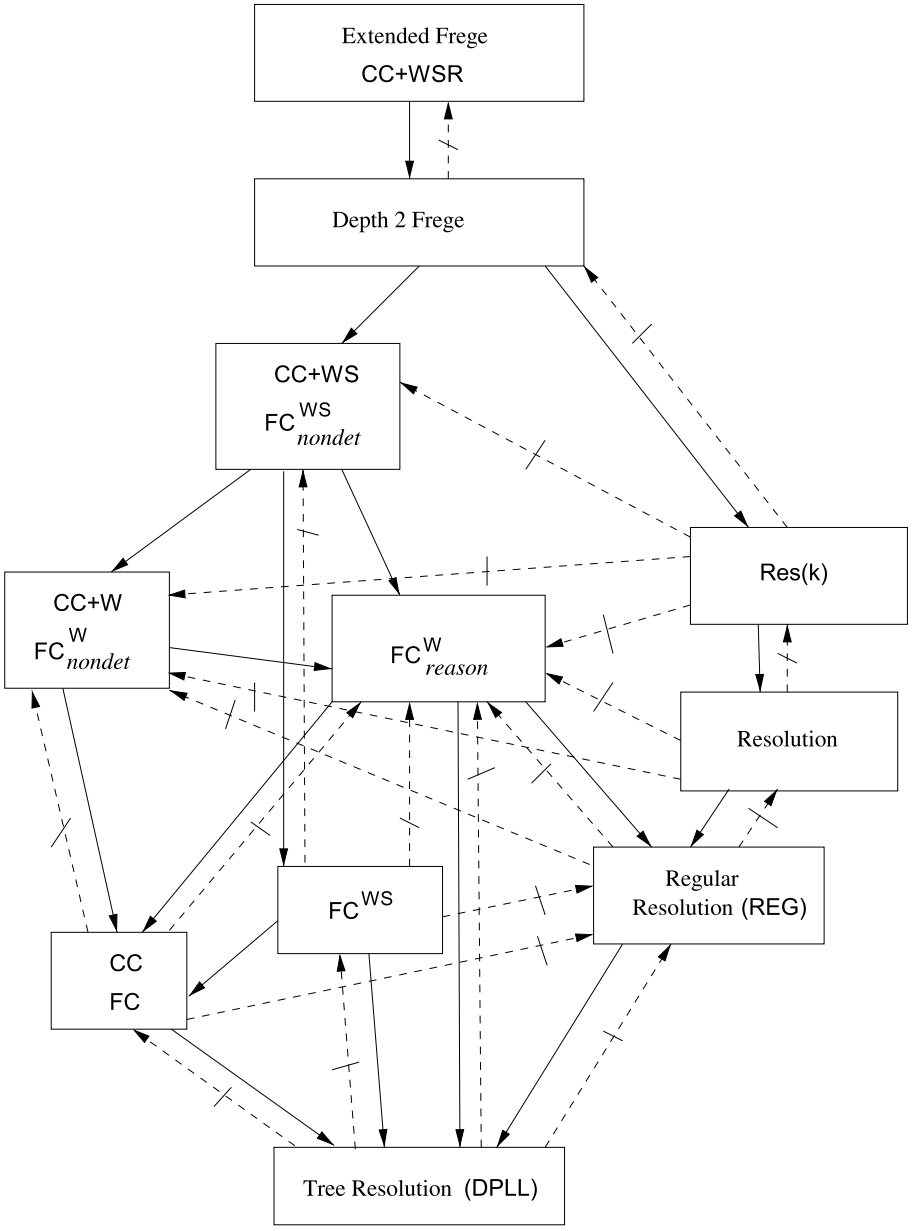


Fig. 8. The relative complexity of caching proof systems. Solid arcs denote p-simulation. Dashed arcs with slashes denote exponential separation.

## 4. THE RELATIVE COMPLEXITY OF CACHING PROOF SYSTEMS

Figure 8 shows the relative complexity of our two new types of caching systems compared with standard proof systems related to resolution. Two proof systems within the same box indicate that they are p-equivalent. An arrow

from proof system $\mathsf{V}_2$ to $\mathsf{V}_1$ indicates that $\mathsf{V}_2$ p-simulates $\mathsf{V}_1$. A dashed arrow with a slash from $\mathsf{V}_1$ to $\mathsf{V}_2$ indicates that $\mathsf{V}_1$ does not p-simulate $\mathsf{V}_2$.

In order to present a more manageable view of the hierarchy of proof systems, we concentrate on the main systems only in Figure 8. In particular, we could have presented more variations of the FC and CC systems, augmented with every possible subset of {W, S}. However, because $W$, and $S$ are tractable, we chose to include either both ($\mathsf{FC}^{\mathsf{WS}}$ and CC+WS) or neither of them (FC and CC). In practice, this makes sense since adding both to the system is nearly as efficient as adding only one. However, when we state our simulations and separations, we will present the weakest possible system necessary for any upper bound, and the strongest possible system for the lower bounds.

### 4.1 Simulations between Proof Systems

Immediately from their definitions, we have the following easy p-simulations.

PROPOSITION 4.1. *Let* $\mathsf{T} \subseteq \{\mathsf{W}, \mathsf{S}\}$*, and let* $\mathsf{T}' \subseteq \mathsf{T}$*. Then we have the following simulations:*

(1.) CC+T *p-simulates* CC+T′.
(2.) $\mathsf{FC}^{\mathsf{T}}_{nondet}$ *p-simulates* $\mathsf{FC}^{\mathsf{T}'}_{nondet}$.
(3.) $\mathsf{FC}^{\mathsf{T}}_{reason}$ *p-simulates* $\mathsf{FC}^{\mathsf{T}'}_{reason}$.
(4.) $\mathsf{FC}^{\mathsf{T}}$ *p-simulates* $\mathsf{FC}^{\mathsf{T}'}$.
(5.) $\mathsf{FC}^{\mathsf{T}}_{nondet}$ *p-simulates* $\mathsf{FC}^{\mathsf{T}}_{reason}$.

It is clear that the basic CC proof system can efficiently simulate the execution of any DPLL algorithm, and thus can p-simulate tree-like resolution proofs. (The axiom and Limited Weakening together simulate the action at the leaves and the Branching rule simulates the action at the internal nodes of the proof.) Also, because FC is a generalization of DPLL, FC p-simulates DPLL. Thus we have the following lemma.

LEMMA 4.2. *Both* CC *and* FC *p-simulate DPLL.*

We now show how the FC and CC systems are related to each other.

THEOREM 4.3. *For any* $\mathsf{T} \subseteq \{\mathsf{W}, \mathsf{S}\}$*,* CC+T *is p-equivalent to* $\mathsf{FC}^{\mathsf{T}}_{nondet}$*. In particular,*

(1.) CC *is p-equivalent to* FC.
(2.) CC+W *is p-equivalent to* $\mathsf{FC}^{\mathsf{W}}_{nondet}$.
(3.) CC+WS *is p-equivalent to* $\mathsf{FC}^{\mathsf{WS}}_{nondet}$.

PROOF. We first do the forward directions: We construct each CC+T proof to consist of the formulas in the cache of the $\mathsf{FC}^{\mathsf{T}}_{nondet}$ execution in the order in which they were added. To show that CC can efficiently simulate FC, observe that in an execution of FC, each recursive call adds precisely one formula to $L$ and each such formula $F$ is derivable either because it contains the empty clause $\Lambda$ and therefore follows from the axiom of CC via one step of Limited Weakening, or as the result of $F|_x$ and $F|_{\overline{x}}$ being in $L$ and therefore follows via

one Branching step. Similarly, CC+W can p-simulate $\mathsf{FC}^{\mathsf{W}}_{nondet}$, and CC+WS can p-simulate $\mathsf{FC}^{\mathsf{WS}}_{nondet}$. If some formula $F'$ was derivable from $F$ by nondeterministic Weakening, then this can be simulated by a Weakening step of CC+W. Similarly, nondeterministic subsumption can be simulated by a Subsumption step of CC+WS. Thus it is left to show that $\mathsf{FC}^{\mathsf{T}}_{nondet}$ can p-simulate CC+T.

Let $F$ be the goal formula for CC+T which will be the input for $\mathsf{FC}^{\mathsf{T}}_{nondet}$. Draw the DAG of inferences in this CC+T proof with edges directed from each formula back to its antecedents. Remove all formulas in the proof that are not reachable from the goal formula $F$; by construction, this is still a CC+T derivation of $F$. We apply induction on the size of CC+T derivations and assume by induction that at each point in a postorder traversal of this DAG, the cache $L$ of $\mathsf{FC}^{\mathsf{T}}_{nondet}$ contains the formulas for all Branching nodes in the DAG that have been fully explored up to this point.

The $\mathsf{FC}^{\mathsf{T}}_{nondet}$ algorithm will follow a depth-first traversal of this DAG and make a recursive call to $\mathsf{FC}^{\mathsf{T}}_{nondet}$ on the input formula, and on the formula for each child of an outdegree two (Branching) node in the DAG. Let $\varphi$ be such a formula. We describe the execution of the recursive call $\mathsf{FC}^{\mathsf{T}}_{nondet}(\varphi, L)$:

Consider the path of outdegree one nodes in the DAG from $\varphi$ to the first node $\psi$ that contains $\Lambda$ or is the result of a branching inference (outdegree 2). (If $\varphi$ already contains $\Lambda$ then this path is empty and $\psi = \varphi|_x$.) This (possibly empty) path contains only inferences in T. It is easy to see that any sequence of Weakening inferences in a CC+W derivation can be simulated by a single instance of the nondeterministic reverse weakening from $\varphi$ to $\psi$ in $\mathsf{FC}^{\mathsf{W}}_{nondet}$, any sequence of Subsumption inferences in the CC+S derivation can be simulated by a single instance of the nondeterministic reverse subsumption from $\varphi$ to $\psi$, and that any interleaved sequence of Weakening and Subsumption inferences in a CC+WS derivation can be simulated by a single instance of nondeterministic reverse weakening followed by nondeterministic reverse subsumption in $\mathsf{FC}^{\mathsf{WS}}_{nondet}$. Thus, in any case, $\mathsf{FC}^{\mathsf{T}}_{nondet}$ can produce the same $\psi$ as in the CC+T derivation.

If the formula $\psi$ contains $\Lambda$ (it is an axiom or follows from $\Lambda$ by Limited Weakening) then the call $\mathsf{FC}^{\mathsf{T}}_{nondet}(\varphi, L)$ will return without finding a satisfying assignment to $\varphi$. If the node with the formula $\psi$ in the DAG is the result of a Branching inference and has been fully explored then by the inductive assumption $\psi$ is in the cache $L$ and the call $\mathsf{FC}^{\mathsf{T}}_{nondet}(\varphi, L)$ will return without finding a satisfying assignment for $\varphi$. Otherwise $\psi$ is the result of Branching inference on some variable $x$ but has not yet been fully explored. We can suppose without loss of generality that the depth-first traversal visits the node labeled $\psi|_x$ before $\psi|_{\overline{x}}$. Then since $\psi$ is not in the cache, the execution of $\mathsf{FC}^{\mathsf{T}}_{nondet}(\varphi, L)$ can choose this literal $x$ and therefore make recursive calls to $\mathsf{FC}^{\mathsf{T}}_{nondet}(\psi|_x, L)$, followed by $\mathsf{FC}^{\mathsf{T}}_{nondet}(\psi|_{\overline{x}}, L)$. By the inductive hypothesis both calls return without finding a satisfying assignment and add formulas to the cache for all fully explored branching nodes below them. Finally, $\mathsf{FC}^{\mathsf{T}}_{nondet}(\varphi, L)$ adds the formula $\psi$ to the cache, finishes exploring the descendants of $\varphi$ and returns without finding a satisfying assignment for $\varphi$. This yields the claimed property for $L$ as a result of this recursive call. The number of recursive calls of $\mathsf{FC}^{\mathsf{T}}_{nondet}$ is at

most the size of the CC+T derivation and each recursive call can be efficiently simulated. □

We prove that both CC and CC+W have the *subformula property* which can be useful for understanding the structure of proofs. It does not seem that CC+WS has the subformula property; one symptom of this is the fact that in a CC+WS proof, it is possible to branch on a variable $x$ more than once along a path in the proof DAG.

*Definition* 4.4. A CNF formula $F$ is a *subformula* of another CNF formula $G$ if every clause of $F$ is contained in some clause of $G$. A CNF refutation system V has the *subformula property* if for any unsatisfiable formula $F$ there is a refutation of $F$ of size at most $s_V(F)$ such that every line is a subformula of $F$.

LEMMA 4.5. CC *and* CC+W *have the subformula property.*

PROOF. Let $\varphi$ be derived from $\varphi_1$ and $\varphi_2$ via a sound inference rule. The rule is *monotone* if for every clause $C$ in $\varphi_1$ or $\varphi_2$, there is a clause $C'$ in $\varphi$ that contains $C$. It is easy to check that all rules of CC and CC+W are monotone, and thus any derivation in CC or CC+W is monotone. Now from this it is easy to see that the subformula property holds. Suppose for sake of contradiction that that we have a CC+W (or CC) refutation of $F$, and further assume that some intermediate formula $G$ is not a subformula of $F$.

Then there is there is some clause $C$ of $F$ that is not contained in any clause of $G$. But this contradicts our monotonicity condition. □

Next we will prove that CC+W has at least the power of regular resolution.

THEOREM 4.6. CC+W *p-simulates regular resolution.*

PROOF. Let $\mathcal{C} = C_1, ..., C_s = \Lambda$ be a regular resolution refutation of $F$. The structure of this refutation can be revealed by viewing the refutation as a directed acyclic graph $P$. Each node in $P$ corresponds to a clause from $\mathcal{C}$; the root node (the node with indegree 0) corresponds to the empty clause $C_s = \Lambda$, and each leaf node (nodes with outdegree 0) corresponds to a clause from $F$. If clause $C_k$ is derived from clauses $C_i$ and $C_j$ in $\mathcal{C}$, then there are directed edges from $C_k$ to $C_i$ and from $C_k$ to $C_j$.

For each clause $C$ in the refutation, define $V'(C)$ to be the set of variables queried at descendants of the node corresponding to $C$ in $P$. By the read-once property of $P$, any variable in $V'(C)$ cannot appear on any path from the root to $C$ in $P$. For each such clause $C$, define $F\#_C$ to be the CNF formula consisting of the clauses of $F|_{\overline{C}}$ having variables only in $V'(C)$.

We will show how to derive the sequence $F\#_{C_1}, \ldots, F\#_{C_s} = F\#_\Lambda$ which will be enough to derive $F$ in one more step since $F$ is (at worst) a weakening of $F\#_\Lambda$.

If $C$ is a clause of $F$ which must be a leaf in the proof, then $F\#_C$ contains the empty clause and we can derive it in two steps using the axiom and Weakening.

Suppose $C = (A \vee B)$ is the resolvent of $(A \vee x)$ and $(B \vee \overline{x})$ in the proof and that we already have derived $F\#_{(A\vee x)}$ and $F\#_{(B\vee\overline{x})}$.

Since every literal in $C = (A \vee B)$ appears on every/some path from the root to the node of $P$ corresponding to $C$, no variable in $A$ or $B$ appears in $V'(A \vee x)$ or in $V'(B \vee \overline{x})$. Therefore $F\#_{(A \vee x)}$ does not contain any variable from $B$ and $F\#_{(B \vee \overline{x})}$ does not contain any variable from $A$. Therefore $F\#_{(A \vee x)}|_{\overline{B}} = F\#_{(A \vee x)}$.

Now every clause of $F\#_{(A \vee x)} = F\#_{(A \vee x)}|_{\overline{B}}$ is a clause of $F|_{\overline{(A \vee B \vee x)}}$ by definition. Furthermore, since $V'(A \vee x)$ is a subset of $V'(C)$, each clause of $F\#_{(A \vee x)}$ is also entirely defined on $V'(C)$. Therefore by one step of Weakening from $F\#_{(A \vee x)}$ we derive the CNF formula consisting of the clauses of $F|_{\overline{(A \vee B \vee x)}} = (F|_{\overline{C}})|_{\overline{x}}$ that only contain variables in $V'(C)$. Similarly by one step of Weakening from $F\#_{(B \vee \overline{x})}$ we can derive the CNF formula consisting of the clauses of $F|_{\overline{(A \vee B \vee \overline{x})}} = (F|_{\overline{C}})|_{x}$ that only contain the variables in $V'(C)$. Finally, using the Branching rule we derive $F\#_C$. □

CC+W is equivalent to $\mathsf{FC}^{\mathsf{W}}_{nondet}$ which does not seem to be particularly implementable. As we will see in Section 4.2, if we consider only the basic $\mathsf{FC}^{\mathsf{T}}$ proof systems we will not be able to match the power of regular resolution. However, when we augment formula caching by having it return the reason for unsatisfiability as well as the mere fact of unsatisfiability, we can still efficiently simulate regular resolution (and much more, as we will see shortly). To do this we first make the following observations about the execution of $\mathsf{FC}^{\mathsf{W}}_{reason}$.

*Definition* 4.7. Define the *dynamic programming DAG* of an execution of $\mathsf{FC}^{\mathsf{W}}_{reason}$ on input $F$ as a (directed acyclic) graph with a node for each recursive call made by $\mathsf{FC}^{\mathsf{W}}_{reason}$. The label of a node associated with a recursive call for formula $F'$ is a pair $(F', J')$. Some nodes will also be tagged with variable names. The graph is built as the algorithm proceeds. Consider the execution of a recursive call on $F'$. We have the following cases:

—If $F'$ contains $\Lambda$ then label the node with $(F', \Lambda)$; the node will have no out-edges and therefore will be a sink in the DAG.

—If Cache-Check was true for this call and a reason $J'$ for the unsatisfiability of $F'$ was found and returned from the cache then label the node $(F', J')$. There must have been some previous recursive call on $F'''$ on which Cache-Add was true that caused $J'$ to be placed in the cache. Add an edge from $(F', J')$ to the node for the recursive call on $F'''$. (It is possible that $F''' = F'$.)

—Otherwise, let $x$ be the literal chosen for branching. Tag the node for $F'$ with this variable. Add an edge (the left edge) to a node for the recursive call on $F'|_x$, finish that recursive call, add an edge (the right edge) to a node for the recursive call on $F'|_{\overline{x}}$, and label the node for this call on $F'$ with the pair $(F', J')$ where $J'$ is the reason returned by the call on input $F'$.

We say that a formula $G$ is a *strengthening* of a formula $H$, written $G \sqsubseteq H$, if and only if $H$ is a weakening of $G$.

LEMMA 4.8. *In the dynamic programming DAG for an execution of* $\mathsf{FC}^{\mathsf{W}}_{reason}(F, \emptyset)$*, if a node* $v$ *is labeled* $(F', J')$ *then:*

(a) $J'$ *is the reason returned on the associated recursive call on* $F'$ *in this execution,*

(b) *if $v$ has outdegree 1 then it points to a node labeled $(F'', J')$ of outdegree 2, and*

(c) *$J'$ is a strengthening of $F'$.*

PROOF. Part (a) is immediate from the definition. Part (b) follows since any node of outdegree 1 points to a node for a call in which a formula was placed in the cache, which only happens at branching nodes.

We prove part (c) by induction starting at the nodes of outdegree 0 in the dynamic programming DAG. Nodes of outdegree 0 have labels $(F', \Lambda)$ such that $F'$ contains $\Lambda$. This clearly satisfies (c). We now have two cases.

The node labeled $(F', J')$ has outdegree 1 and results from a cache hit that returned $J'$. Since this is a cache hit, $F'$ must be a weakening of $J'$ so (c) holds.

The node labeled $(F', J')$ has outdegree 2 and results from combining the recursive calls for $F'|_x$ and $F'|_{\overline{x}}$. This node is tagged with the variable in $x$ and the two nodes it points to are labeled $(F'|_x, G')$ and $(F'|_{\overline{x}}, H')$ for some $G'$ and $H'$. We apply the inductive hypothesis to each of these nodes. In particular, all clauses in $G'$ are in $F'|_x$ and all clauses in $H'$ are in $F'|_{\overline{x}}$. By construction, if a clause $C$ occurs in $G'$ but not $H'$ then $C$ is in $F'|_x$ and $(C \vee \overline{x})$ must be a clause in $F'$. Similarly if a clause $C$ occurs in $H'$ but not $G'$ then $(C \vee x)$ must be a clause in $F'$. If $C$ occurs in both $G'$ and $H'$ then it occurs in both $F'|_x$ and $F'|_{\overline{x}}$. Therefore either $C$ occurs in $F'$ or both $(C \vee x)$ and $(C \vee \overline{x})$ occur in $F'$. Therefore by the construction of $J'$ in the $\mathsf{FC}^{\mathsf{W}}_{reason}$ code, every clause of $J'$ is in $F'$ and (c) holds. □

THEOREM 4.9. *$\mathsf{FC}^{\mathsf{W}}_{reason}$ p-simulates regular resolution.*

PROOF. We follow the general pattern of the proof of Theorem 4.6. We describe an execution of $\mathsf{FC}^{\mathsf{W}}_{reason}$ on input $F$ so that the dynamic programming DAG of the $\mathsf{FC}^{\mathsf{W}}_{reason}$ execution is essentially the same as the regular resolution DAG refuting $F$ (and is constructed as a depth-first search of that DAG). As in the proof of Theorem 4.6, in a regular resolution proof of $F$, $V(C)$ consists of the set of variables that are queried below the node corresponding to clause $C$, and $F\#_C$ consists of those clauses of $F|_{\overline{C}}$ having variables only in $V(C)$. (By the read-once property of the proof, the set $V(C)$ is disjoint from the set of variables queried along the path from the root to $C$.)

We will prove inductively that the branching nodes of the dynamic programming DAG are in a 1-1 correspondence with the nonsink nodes of the regular resolution DAG such that a node labeled by clause $C$ in the regular resolution DAG corresponds to a branching node with label $(F|_{\overline{D}}, J_D)$ such that the following invariant properties hold.

(1) $D$ contains $C$ ($C \subseteq D$);

(2) $D$ is disjoint from $V'(C)$ (thus, by (1) and (2), $F\#_C \sqsubseteq F|_{\overline{D}}$; that is, $F\#_C$ is a strengthening of $F|_{\overline{D}}$);

(3) $J_D \sqsubseteq F\#_C$; that is, $J_D$ is a strengthening of $F\#_C$;

(4) $J_D$ is in the cache when the call for that node completes.

Since by Lemma 4.8(b) nodes of outdegree 1 can only occur singly between pairs of outdegree 2 nodes, this will show that the size of the dynamic programming DAG will be linear in the size of the regular resolution DAG. The theorem will follow since the running time of $\mathsf{FC}^{\mathsf{W}}_{reason}$ is polynomial in the size of its dynamic programming DAG and the size of the input formula since it has one node for each recursive call.

To define the execution, we follow the depth-first search of the regular resolution DAG from the root labeled $\Lambda$. In this execution *Cache-Add* will always be true. We express this as an induction based on the completion times for the nodes in the depth-first search of the regular resolution DAG. That is, we show that if (1)–(4) are true for all nodes in the DAG completed prior to visiting node $v$ and if (1) and (2) are true for $v$ when $v$ is first visited then (1)–(4) will be true for $v$ and all nodes completed at the time that $v$ is completed. Observe that the initial call of $\mathsf{FC}^{\mathsf{W}}_{reason}$ is on the formula $F|_{\overline{\Lambda}} = F$ as required for the roots to correspond. Therefore, since (1) and (2) hold for the root which is completed last, the invariants follow.

Consider a node labeled $C$ for which the corresponding recursive call on $F|_{\overline{D}}$ has been made by $\mathsf{FC}^{\mathsf{W}}_{reason}$. If $C$ is a clause of $F$ then the corresponding $D$ from (1) satisfies $C \subseteq D$ so $F|_{\overline{D}}$ will contain $\Lambda$, the corresponding node will be a sink in the dynamic programming DAG and the execution will return. If $C$ is a derived clause of the regular resolution proof then $C = (A \vee B)$ is the resolvent of some pair of clauses $(A \vee x)$ and $(B \vee \overline{x})$ where $(B \vee \overline{x})$ is the first of the two children of $C$ to be explored in the depth-first search of the regular resolution DAG. In the execution of $\mathsf{FC}^{\mathsf{W}}_{reason}$ on input $F|_{\overline{D}}$, we select *Cache-Check* to be false, $x$ to be the literal chosen, and *Cache-Add* to be true. (We can still choose $x$ since $x \in V'(C)$ and by property (2), $V'(C)$ is disjoint from the variables in $D$, and thus $x$ does not occur in $D$.)

$\mathsf{FC}^{\mathsf{W}}_{reason}$ will first make a recursive call on $(F|_{\overline{D}})|_x$. By the argument in the proof of Theorem 4.6, $F\#_{(B\vee\overline{x})} \sqsubseteq (F|_{\overline{D}})|_x$. Since $D$ contains $C$ and the variables in $D$ are disjoint from $V'(C)$, $(F|_{\overline{D}})|_x$ contains all clauses in $(F|_{\overline{C}})|_x$ on $V'(C)$. Therefore, since $V'(B \vee \overline{x})$ is a subset of $V'(C)$, $F\#_{(B\vee\overline{x})} \sqsubseteq (F|_{\overline{D}})|_x$. Also, any variable in $(D \vee \overline{x})$ but not $(B \vee \overline{x})$ is disjoint from $V'(B \vee \overline{x})$.

If $(B \vee \overline{x})$ is a clause in the proof that has not yet been explored then we can apply the preceding argument inductively for $(B \vee \overline{x})$ and the call on $F|_{\overline{(D\vee\overline{x})}} = (F|_{\overline{D}})|_x$ to return some $J_{D'}$ such that $J_{D'} \sqsubseteq F\#_{(B\vee\overline{x})} \sqsubseteq (F|_{\overline{D}})|_x$.

If $(B \vee \overline{x})$ is a clause in the proof that has previously been explored then by the inductive hypothesis that node is labeled by a pair $(F|_{\overline{D'}}, J_{D'})$ such that $D'$ is disjoint from $V'(B \vee \overline{x})$, $J_{\overline{D'}} \sqsubseteq F\#_{(B\vee\overline{x})}$, and $J_{\overline{D'}}$ is in the cache. Therefore on the call $(F|_{\overline{D}})|_x$ we select *Cache-Check* to be true. In this case $J_{D'} \sqsubseteq F|_{\overline{D}}|_x$, so we select $J_{D'}$ to be returned from the cache as the reason for the unsatisfiability of $(F|_{\overline{D}})|_x$.

After the return from the call on $(F|_{\overline{D}})|_x$, the same argument is applied to the other call on $(F|_{\overline{D}})|_{\overline{x}}$ to derive that it returns a $J_{D''}$ such that $J_{D''} \sqsubseteq F\#_{(A\vee x)} \sqsubseteq (F|_{\overline{D}})|_x$.

Thus the reasons $J_{D'}$ and $J_{D''}$ returned from the two recursive calls satisfy $J_{D'} \sqsubseteq F\#_{(B\vee\overline{x})}$ and $J_{D''} \sqsubseteq F\#_{(A\vee x)}$. Then, by construction, the clauses of the

formula $J_D = J$ that is returned from the call on $F|_{\overline{D}}$ are defined on $V'(C) = \{x\} \cup V'(A \vee x) \cup V'(B \vee \overline{x})$ and by Lemma 4.8(c) they are contained in $F|_{\overline{D}}$. Since $D$ is an extension of $C$ that is disjoint from $V'(C)$, $F\#_C$ is precisely the set of all clauses of $F|_{\overline{D}}$ that are defined on $V'(C)$ and thus $J_D \sqsubseteq F\#_C$ as required. Since *Cache-Add* is true, $J_D$ will be in the cache when this call returns. □

The following lemma shows that $\mathcal{F}_2$ can p-simulate CC+WS, and therefore all of the caching systems introduced so far can be p-simulated by $\mathcal{F}_2$.

LEMMA 4.10. *$\mathcal{F}_2$ p-simulates* CC+WS.

PROOF. We want to show that $\mathcal{F}_2$ can p-simulate CC+WS. Technically speaking, the axiom $\Lambda$ and any clause containing it cannot be derived because it is not representable. Still, we can show inductively how to efficiently convert any CC+WS refutation of a CNF formula that does not contain $\Lambda$. Weakening is equivalent to R1', and Subsumption is equivalent to R2'. We first show the base case that any clause in the CC+WS refutation not containing $\Lambda$ whose antecedent(s) contain $\Lambda$ can be derived. Observe that removing $\Lambda$ in the CC+WS refutation requires a Branching rule that creates clauses $x \wedge \neg x$, among others, which can be derived in $\mathcal{F}_2$. The clauses $x \wedge \neg x$ can be augmented by Weakening (R1') to produce the corresponding formula in the CC+WS refutation. Since we have the base case as well as Weakening and Subsumption it is left to show how to simulate Branching. We want to show how to derive some formula $F$ from $F|_x$ and $F|_{\overline{x}}$ in $\mathcal{F}_2$. Assume that $F$ has the following form.

$$(x \vee D_1) \wedge \ldots \wedge (x \vee D_j) \wedge (\overline{x} \vee E_1) \wedge \ldots \wedge (\overline{x} \vee E_k) \wedge G.$$

Then $F|_{\overline{x}}$ is equal to $(D_1) \wedge \ldots \wedge (D_j) \wedge G$, and $F|_x$ is equal to $(E_1) \wedge \ldots \wedge (E_k) \wedge G$.

From $D_1 \wedge D_2 \wedge \ldots \wedge D_j$ and $(x \wedge \overline{x})$, derive $(D_1 \vee x) \wedge \ldots \wedge (D_j \vee x) \wedge (\overline{x}) \wedge G$, by repeated applications of R1' and R3'. Similarly, from $E_1 \wedge E_2 \wedge \ldots \wedge E_k$ and $(x \wedge \overline{x})$, derive $(E_1 \vee \overline{x}) \wedge \ldots \wedge (E_k \vee \overline{x}) \wedge (x) \wedge G$. Now use R2' and R4' to derive $F$ as desired. □

## 4.2 Separations between Proof Systems

In this section, we will show that DPLL cannot p-simulate even the most basic caching systems, FC and CC. We give CNF examples where $\mathsf{FC}^{\mathsf{W}}_{reason}$ has polynomial size refutations but that are known to require exponential size resolution and Res($k$) refutations.

The idea behind most of our lower bounds is as follows. Suppose that we want to show that some resolution-like system $\mathcal{R}$ cannot p-simulate a particular caching system, call it $\mathcal{C}$. We will begin with a CNF formula $F$ that has a small proof in $\mathcal{R}$, but such that if we replace each variable in $F$ by a small conjunction of variables and distribute to again obtain a CNF formula, then the resulting formula, $F'$, now requires large $\mathcal{R}$-proofs. On the other hand, we will show that the caching system $\mathcal{C}$ can prove $F'$ efficiently whenever it can prove $F$ efficiently. Thus if $\mathcal{C}$ can efficiently prove $F$, then it will follow that $F'$ is our formula that has short $\mathcal{C}$-proofs, but that requires large $\mathcal{R}$-proofs. We proceed formally as follows.

*Definition* 4.11. The *size* (or *width*) of clause $C$ is the number of literals in $C$. The *clause-width* (or simply *width*) of a CNF formula $F$ is the maximum width of any of its clauses.

*Definition* 4.12. Let $F$ be a CNF formula. We can define a new formula $F^{(\wedge k)}$ in variables $\{z_{i,j} : i \in [n], j \in [k]\}$ by replacing every clause $C \in F$ by a conjunction of clauses corresponding to $C$ with the substitution $x_i \leftarrow z_{i,1} \wedge \cdots \wedge z_{i,k}$ and distributing the result to form clauses. That is, if $P$ and $N$ are the indices of variables occurring positively and negatively in $C$ then $C$ is replaced by $\bigwedge_{(j_1,\ldots j_{|P|})\in[k]^{|P|}} \left(\bigvee_{i\in P} z_{i,j_i} \vee \bigvee_{i\in N}\bigvee_{j\in[k]} \overline{z_{i,j}}\right)$. Note that if $C$ has at most $d$ positive literals then it is replaced by at most $k^d$ clauses each of size at most $dk$. Thus if $F$ has at most $d$ positive literals per clause then $F^{(\wedge k)}$ has size at most $k^{d+1}$ times the size of $F$.

*Definition* 4.13. Let $\pi$ be a partial assignment to the $x$ variables (which we identify with the sequence of literals on those variables made true by the assignment). We say that a partial assignment $\hat{\pi}$ to the $z$ variables is *equivalent* to $\pi$ if and only if for every $i$:

(i) if $x_i$ is in $\pi$ then $z_{i,j}$ is in $\hat{\pi}$ for all $j \in [k]$;
(ii) if $\overline{x}_i$ is in $\pi$ then there is some $\overline{z_{i,j}}$ in $\hat{\pi}$;
(iii) if neither $x_i$ not $\overline{x}_i$ is in $\pi$ then none of the $z_{i,j}$ nor $\overline{z_{i,j}}$ is in $\hat{\pi}$.

The following lemma follows from the definitions.

LEMMA 4.14. *Let $\pi$ be a partial assignment to the $x$ variables on which CNF formula $F$ is defined, and let $\hat{\pi}$ be an equivalent assignment to the $z$ variables. Then $(F^{(\wedge k)})|_{\hat{\pi}} = (F|_\pi)^{(\wedge k)}$.*

LEMMA 4.15. *If* V *is any of the systems* CC, CC+W, FC, $\mathsf{FC}^{\mathsf{W}}$, *or* $\mathsf{FC}^{\mathsf{W}}_{reason}$, *then for any unsatisfiable CNF formula $F$ with at most $d$ positive literals per clause, then $s_{\mathsf{V}}(F^{(\wedge k)}) \leq 2k^{d+2} \cdot s_{\mathsf{V}}(F)$.*

PROOF. CC and CC+W have the subformula property by Lemma 4.5, $\mathsf{FC}^{\mathsf{W}}$ has the subformula property by construction and $\mathsf{FC}^{\mathsf{W}}_{reason}$ has the subformula property by Lemma 4.8(c). Since the substitution $F^{(\wedge k)}$ increases the size of each subformula of $F$ by at most a $k^{d+1}$ factor it suffices to prove an upper bound on the number of clauses in a refutation of $F^{(\wedge k)}$ as a function of that of $F$.

First, given a CC or CC+W refutation $\Pi$ of $F$ of length $s$ we show how to derive all clauses of $\Pi^{(\wedge k)}$ using at most $sk$ inference steps. Consider the rules used in the course of the refutation $\Pi$.

(1) Clearly $\Lambda^{(\wedge k)} = \Lambda$.
(2) If the inference rule in $\Pi$ is Weakening $\varphi \dashv \varphi \wedge \psi$ and we have already $\varphi^{(\wedge k)}$ then we get $\varphi^{(\wedge k)} \dashv \varphi^{(\wedge k)} \wedge \psi^{(\wedge k)}$ also by Weakening and the latter is $(\varphi \wedge \psi)^{(\wedge k)}$ by definition. Further, if the Weakening inference in $\Pi$ is limited then the same will hold true in $\Pi^{(\wedge k)}$.

(3) Suppose that clause $\varphi \in \Pi$ follows from $\varphi|_x$ and $\varphi|_{\overline{x}}$ using Branching and the substitution is $x = z_1 \wedge \ldots \wedge z_k$. (We have dropped the indices $i$ from both the $x$ and $z$ variables for convenience.) For $j \in [k]$, let $F_j = \varphi^{(\wedge k)}|_{z_1 \ldots z_j}$ and $G_j = \varphi^{(\wedge k)}|_{z_1 \ldots z_{j-1}\overline{z_j}}$. As before, $F_k = \varphi^{(\wedge k)}|_{z_1 \ldots z_k} = (\varphi|_x)^{(\wedge k)}$. Furthermore, as before $G = (\varphi|_{\overline{x}})^{(\wedge k)} = \varphi^{(\wedge k)}|_{\overline{z_j}}$ for any $j \in [k]$. Since $G$ contains no occurrences of $z_1, \ldots, z_k$ for $j \in [k]$ we can also write $G = G|_{z_1 \ldots z_{j-1}} = \varphi^{(\wedge k)}|_{\overline{z_j} z_1 \ldots z_{j-1}} = G_j$. We wish to derive $\varphi^{(\wedge k)}$ from $F_k$ and $G = G_1 = \ldots = G_k$. To do this we apply the branching rule $k$ times, deriving $F_{k-1}$ from $F_k$ and $G_k$ using variable $z_k$, $F_{k-2}$ from $G_{k-1}$ and $F_{k-1}$ using variable $z_{k-1}$, etc., until finally we obtain the desired clause using the Branching rule applied to $F_1$ and $G_1$.

Next we will show the same result for $\mathsf{FC}^{\mathsf{W}}_{reason}$. The argument for $\mathsf{FC}^{\mathsf{W}}$ is a simplification of this proof and the result for FC follows because it is equivalent to CC. Given a refutation of a formula $F$ in $\mathsf{FC}^{\mathsf{W}}_{reason}$, we show how to obtain an $\mathsf{FC}^{\mathsf{W}}_{reason}$ refutation of $F^{(\wedge k)}$ of size at most $O(k)$ times that of $F$ by replacing each branch on a variable $x$ of $F$ by a sequence of branches on the variables $z_j$ for $j \in [k]$. (Again we drop the indices $i$ on the $x$ and $z$ variables for convenience.) More precisely, let $F$ be a CNF formula, and let $T$ be the dynamic programming DAG explored by $\mathsf{FC}^{\mathsf{W}}_{reason}$ as it is refuting $F$. $T^{(\wedge k)}$ will denote the corresponding dynamic programming DAG that we show can be created by $\mathsf{FC}^{\mathsf{W}}_{reason}$ as it is refuting $F^{(\wedge k)}$.

We will define an execution creating a $T^{(\wedge k)}$ so that for any partial assignment $\pi$ to the $x$ variables defining a node $v(\pi)$ in $T$ corresponding to a recursive call on $F|_\pi$, there is an equivalent assignment $z(\pi)$ defining a node $\hat{v}(\pi)$ in $T^{(\wedge k)}$ such that for every formula $G$ cached in $T$ when exploring node $v(\pi)$ the corresponding formula $G^{(\wedge k)}$ is cached in $T^{(\wedge k)}$ and if $J$ is the reason returned at $v(\pi)$, $J^{(\wedge k)}$ is returned at $\hat{v}(\pi)$. We prove this by induction over the execution that yields $T$.

Let $\pi$ be an assignment that corresponds to a node in $T$. We define an equivalent assignment $z(\pi)$ that will correspond to a node of $T^{(\wedge k)}$ recursively as follows.

—If $\pi$ is the empty assignment then $z(\pi)$ is also empty.
—If $\pi$ corresponds to node $v$ in $T$ with left child corresponding to $\pi x$ and right child corresponding to $\pi\overline{x}$ then $z(\pi x) = z(\pi)z_1 \cdots z_k$ and $z(\pi\overline{x}) = z(\pi)z_1 \cdots z_{k-1}\overline{z}_k$.
—If $\pi$ corresponds to node $v$ in $T$ with left child corresponding to $\pi\overline{x}$ and right child corresponding to $\pi x$ then $z(\pi\overline{x}) = z(\pi)\overline{z}_1$ and $z(\pi x) = z(\pi)z_1 \cdots z_k$.

By the previous definition $z(\pi)$ is equivalent to $\pi$. The node $\hat{v}(\pi)$ will be a node in $T^{(\wedge k)}$ that corresponds to $z(\pi)$.

Assume that the inductive hypothesis holds for all nodes whose execution completed before that of $v = v(\pi)$ in the execution defining $T$ (i.e., the nodes which precede $v$ in the postorder traversal of $T$) where $\pi$ is a partial assignment to the $x$ variables. If $v$ has outdegree 0 in $T$ then $\Lambda$ is in $F|_\pi$ and will also be in $F^{(\wedge k)}|_{z(\pi)} = (F|_\pi)^{(\wedge k)}$ so both calls return $\Lambda$. If $v$ has outdegree 1 in $T$ then it corresponds to a cache hit and some strengthening $J$ of $F|_\pi$ was found in the

cache by $\mathsf{FC}^{\mathsf{W}}_{reason}$ for the call on $F|_\pi$. By the inductive hypothesis, $J^{(\wedge k)}$ will be in the cache for the corresponding call on $F^{(\wedge k)}|_{z(\pi)}$ and will be a strengthening of $F^{(\wedge k)}|_{z(\pi)}$. In this call select *Cache-Check* to be true and select $J^{(\wedge k)}$ to be returned from the cache. It remains to consider what happens when $v$ has outdegree 2. We have two cases: the left child corresponds to a recursive call on $F|_{\pi x}$ and right child corresponds to a recursive call on $F|_{\pi\overline{x}}$, or vice versa. In the execution creating $T^{(\wedge k)}$ of $\mathsf{FC}^{\mathsf{W}}_{reason}$ the query of x at node $v$ will be replaced by a sequence of queries to the variables $z_1, \ldots, z_k$ in order.

First, assume that $x$ is the left (first) branch and $\overline{x}$ is the right (second) branch in $T$ starting at $v$, In $T^{(\wedge k)}$ each positive literal $z_j$ will be assigned before the corresponding negative literal is tried and thus there will be a subtree in the DAG $T^{(\wedge k)}$ of $k+1$ leaves with a long left branch corresponding to the assignment $z(\pi)z_1 \cdots z_k$ and a series of short right branches corresponding to assignments $z(\pi)z_1 \cdots z_{j-1}\overline{z}_j$ for $j \in [k]$. In particular, $T^{(\wedge k)}$ contains the node $\hat{v}(\pi x)$, where $\hat{v}(\pi x)$ is the node in $T^{(\wedge k)}$ corresponding to the assignment $z(\pi x) = z(\pi)z_1 \cdots z_k$ as defined earlier, and similarly $T^{(\wedge k)}$ contains the node $\hat{v}(\pi\overline{x})$ corresponding to the assignment $z(\pi\overline{x}) = z(\pi)z_1 \cdots z_{k-1}\overline{z}_k$.

By definition of $\mathsf{FC}^{\mathsf{W}}_{reason}$, some $G$ is the reason returned at node $v(\pi x)$ and some $H$ is the reason returned at node $v(\pi\overline{x})$ where, by Lemma 4.8, $G$ and $H$ are strengthenings of $F|_{\pi x}$ and $F|_{\pi\overline{x}}$, respectively. In completing the execution for node $v$, the formula

$$J = \bigwedge_{C \in F|_\pi \cap G \cap H} C \wedge \bigwedge_{C \in (G \cap H) \setminus F|_\pi} (x \vee C)(\overline{x} \vee C) \wedge \bigwedge_{C \in G \setminus H} (\overline{x} \vee C) \wedge \bigwedge_{C \in H \setminus G} (x \vee C)$$

is returned. We want to show that $J^{(\wedge k)}$ is returned at node $\hat{v}(\pi)$ in $T^{(\wedge k)}$.

By the induction hypothesis, $G^{(\wedge k)}$ is returned at node $\hat{v}(\pi x)$ which corresponds to assignment $z(\pi x) = z(\pi)z_1 \cdots z_k$. Similarly, $H^{(\wedge k)}$ is returned at node $\hat{v}(\pi\overline{x})$ which corresponds to assignment $z(\pi\overline{x}) = z(\pi)z_1 \cdots z_{k-1}\overline{z}_k$. By the properties of $G$ and $H$, $G^{(\wedge k)}$ and $H^{(\wedge k)}$ are strengthenings of $F^{(\wedge k)}|_{z(\pi x)}$ and $F^{(\wedge k)}|_{z(\pi\overline{x})}$ respectively. Without loss of generality we can assume that these were added to the cache as well.

By the preceding lemma, $(F|_{\pi x})^{(\wedge k)} = F^{(\wedge k)}|_{z(\pi x)}$ and $(F|_{\pi\overline{x}})^{(\wedge k)} = F^{(\wedge k)}|_{z(\pi\overline{x})}$. For all subsequently considered assignments $z(\pi)z_1 \cdots z_{j-1}\overline{z}_j$ for $j < k$, observe that $F^{(\wedge k)}|_{z(\pi)z_1 \cdot z_{j-1}\overline{z}_j}$ contains all clauses of the formula $F^{(\wedge k)}|_{z(\pi\overline{x})}$. (If there is a clause $A$ in $F^{(\wedge k)}|_{z(\pi\overline{x})}$ that was shortened from a clause in $F^{(\wedge k)}|_{z(\pi)}$ then $(A \vee z_j)$ is in $F^{(\wedge k)}|_{z(\pi)}$ for every $j \le k$ and thus $A$ is in $F^{(\wedge k)}|_{z(\pi)z_1 \cdot z_{j-1}\overline{z}_j}$.) Thus at the call corresponding to $z(\pi)z_1 \cdots z_{j-1}\overline{z}_j$, we select *Cache-Check* to be true and obtain a cache hit from the reason $H^{(\wedge k)}$ so these nodes of $T^{(\wedge k)}$ result in immediate contradictions for $\mathsf{FC}^{\mathsf{W}}_{reason}$. We do not bother to cache the intermediate reasons returned at these nodes until the computation returns to node $\hat{v}(\pi)$. By the preceding lemma, $G^{(\wedge k)} = (J|_x)^{(\wedge k)} = J^{(\wedge k)}|_{z(\pi x)}$, and similarly $H^{(\wedge k)} = (J|_{\overline{x}})^{(\wedge k)} = J^{(\wedge k)}|_{z(\pi\overline{x})}$. It follows that $J^{(\wedge k)}$ is returned at node $\hat{v}(\pi)$ since the variables $z_j$ will be added to the returned reasons up the tree from nodes $\hat{v}(\pi x)$ and $\hat{v}(\pi\overline{x})$ to $\hat{v}(\pi)$ to exactly mimic the result of the substitution of $z_1 \wedge \ldots \wedge z_k$ for $x$.

The second case to consider is when $\pi\overline{x}$ corresponds to the left child of $v$ and $\pi x$ corresponds to the right child of $v$ in $T$. In this case the proof proceeds in

much the same way, except that now the subtree of $T^{(\wedge k)}$ has one long right branch corresponding to the assignment $z(\pi)z_1 \cdots z_k$ and $k$ left branches corresponding to assignments $z(\pi)z_1 \cdots z_{j-1}\overline{z}_j$ for $j \in [k]$, for a total of $k+1$ leaves. In this case, the leaf $\hat{v}(\pi\overline{x})$ corresponding to assignment $z(\pi\overline{x}) = z(\pi)\overline{z}_1$ will be traversed first and its returned reason will cause cache hits for leaves with assignments $z(\pi)z_1 \cdots z_{j-1}\overline{z}_j$ for $j > 1$. The only other leaf that will be explored is the leaf $\hat{v}(\pi x)$ corresponding to assignment $z(\pi x)$. The remaining reasoning is completely analogous to the first case. The overall result follows by induction. □

COROLLARY 4.16. *Let $\{F\}$ be a family of unsatisfiable CNF formulas with at most d positive literals per clause where $k^d$ is $n^{O(1)}$.*

—*If $\{F\}$ has polynomial-size DPLL proofs then $\{F^{(\wedge k)}\}$ has polynomial-size* CC *proofs.*

—*If $\{F\}$ has polynomial-size regular resolution proofs then $\{F^{(\wedge k)}\}$ has polynomial-size* CC+W *proofs.*

We first use this corollary to show that CC can be exponentially more powerful than DPLL. Ben-Sasson et al. [2000], generalizing a construction of Bonet et al. [2000], defined certain *graph-pebbling tautologies* $Peb_{G,S,T}$ to separate tree-like from regular resolution.

*Definition* 4.17. Let $G = (V, E)$ be a directed, acyclic graph over vertices $V = \{x_1, \ldots, x_n\}$, and $m$ edges $E$. Assume that $G$ is connected and every nonsource vertex has indegree 2. Let $S$, $T$ be disjoint subsets of $V$. Then $P_{G,S,T}$ is the following CNF formula. The underlying variables are $x_i$, $i \in [n]$. There are three types of clauses: (i) The source clauses, $(x_i)$, for all $x_i \in S$, state that every variable corresponding to a source vertex can be pebbled; (ii) For each nonsource vertex $x_k$ with predecessors $x_i$ and $x_j$, we have a clause $(\neg x_i \vee \neg x_j \vee x_k)$ saying that if both predecessors of $x_k$ are pebbled then $x_k$ can be pebbled; (iii) For every sink vertex $x_j \in T$ we have a clause $(\neg x_j)$ asserting that vertex $x_j$ cannot be pebbled.

Earlier we defined a formula $F^{(\wedge k)}$ from $F$. In an analogous way we can define a formula $F^{(\vee k)}$ from $F$. In particular, we define the "2ORification" of $P_{G,S,T}$ as follows.

*Definition* 4.18. Let $G = (V, E)$ be a directed acyclic graph on $n$ vertices, and let $P_{G,S,T}$ be the CNF formula as defined earlier. Then the 2ORification of $P_{G,S,T}$, $P^{(\vee 2)}$, which we will denote by $Peb_{G,S,T}$ is as follows. The underlying variables are $x_i^a$, $x_i^b$, $i \in [n]$. The clauses are as follows: (i) The source clauses $(x_i^a \vee x_i^b)$ for all $x_i \in S$; (ii) For every nonsource vertex $x_k$ with predecessors $x_i$ and $x_j$, we have the clauses $(\neg x_i^a \vee \neg x_j^a \vee x_k^a \vee x_k^b)$, $(\neg x_i^a \vee \neg x_j^b \vee x_k^a \vee x_k^b)$, $(\neg x_i^b \vee \neg x_j^a \vee x_k^a \vee x_k^b)$, $(\neg x_i^b \vee \neg x_j^b \vee x_k^a \vee x_k^b)$; (iii) The sink clauses $(\neg x_j^a)$, $(\neg x_j^b)$ for all $x_j \in T$.

LEMMA 4.19. *Given a directed acyclic graph $G$ of indegree 2 with $m$ edges and subsets $S$ and $T$ of its vertices, if $Peb_{G,S,T}$ is unsatisfiable then $s_{\mathsf{CC}}(Peb_{G,S,T}) = O(m)$.*

PROOF. (Sketch) Recall that the negation of $P_{G,S,T}$ is a tautology asserting that: (1) all nodes in $S$ can be pebbled; (2) if both predecessors of a node can be pebbled, then so can the node itself; and (3) no node in $T$ can be pebbled. Recall that $Peb_{G,S,T} = P^{(\vee 2)}_{G,S,T}$. The formula $P_{G,S,T}$ can be proved unsatisfiable in a linear number of steps by unit propagation following a topological sort from $S$ to $T$. Therefore it follows immediately in CC. Although this is an $(\vee 2)$ substitution, by negating variables and using the closure property of CC under disjoint $(\wedge 2)$ substitution of Lemma 4.15 and the fact that each clause of $P_{G,S,T}$ has constant size, $Peb_{G,S,T}$ also has a linear size proof in CC. □

THEOREM 4.20. *DPLL cannot p-simulate* FC *nor* CC.

PROOF. Ben-Sasson et al. [2000] show that for suitable choices of directed acyclic graphs $G$ with $O(n)$ edges, and sets $S$ and $T$, the tree-like resolution complexity of $Peb_{G,S,T}$ is $2^{\Omega(n/\log n)}$. This combined with Lemma 4.19 proves that DPLL cannot p-simulate CC (nor FC). □

We now use Corollary 4.16 together with results of Segerlind et al. [2002] to separate the CC+W proof system from Res($k$) for any constant $k$.

In order to separate Res($k+1$) from Res($k$), Segerlind et al. [2002] define an unsatisfiable CNF formula $GOP(G)$ for any undirected graph $G$ (describing the *graph ordering principle* on $G$) as follows.

*Definition* 4.21. Let $G = (V, E)$ be an undirected graph with $|V| = n$. The graph ordering principle defined on $G$ is denoted by $GOP(G)$. There are $n(n-1)$ underlying variables $x_{i,j}$ for all $i, j \leq n, i \neq j$, which are intended to describe a transitive, irreflesiven antisymmetric relation on the vertices of $G$. The clauses express the (false) fact that in any such relation, there is no locally minimal vertex in $G$: (i) The clauses $(\overline{x_{i,j}} \vee \overline{x_{j,i}})$ represent antisymmetry; (ii) the clauses $(\overline{x_{i,j}} \vee \overline{x_{j,k}} \vee x_{i,k})$ represent transitivity for all distinct $i, j, k \in [n]$; (iii) finally for each $j \in [n]$ we have the clause $\bigvee_{(i,j)\in E} x_{i,j}$, expressing that $j$ is not minimal.

The following theorem states that although $GOP(G)$ always has polynomial-size regular resolution refutations, there is an infinite family of graphs $G$ such that for any constant $k$, $GOP(G)^{(\wedge k+1)}$ requires exponential-size Res($k$) refutations.

THEOREM 4.22 [SEGERLIND ET AL. 2002]. *For any positive integer k, there are constants $c > 0$ and $\epsilon_k > 0$, and an infinite family of graphs $\{G\}$ such that $GOP(G)$ has regular resolution refutations of size $O(n^c)$ where $n = n(G)$, and $GOP(G)^{(\wedge k)}$ has* Res($k$) *refutations of size $O(n^c)$, but $GOP(G)^{(\wedge k+1)}$ requires* Res($k$) *refutations of size $2^{\Omega(n^{\epsilon_k})}$.*

THEOREM 4.23. *For any positive integer k, there are formulas with polynomial-size* CC+W *refutations and polynomial-size* $\mathsf{FC}^{\mathsf{W}}_{reason}$ *refutations that require exponential-size* Res($k$) *refutations.*

PROOF. Consider the family of polynomial-size formulas $GOP(G)^{(\wedge k+1)}$. By Theorem 4.22, the formulas $GOP(G)$ have polynomial-size regular resolution refutations. Also, in Theorem 4.22, since $GOP(G)^{(\wedge k+1)}$ has Res($k+1$) refutations of size $O(n^c)$, $(k+1)^d$ is polynomial in $n = n(G)$ where $d$ is the maximum

degree of $G$. Therefore, by Corollary 4.16, the formulas $GOP(G)^{(\wedge k+1)}$ have polynomial-size CC+W refutations and polynomial-size $\mathsf{FC}^{\mathsf{W}}_{reason}$ refutations. On the other hand by Theorem 4.22 they require exponential-size $\mathsf{Res}(k)$ refutations. □

We will now show that the returned reasons are essential to the good properties of $\mathsf{FC}^{\mathsf{W}}_{reason}$ by showing that $\mathsf{FC}^{\mathsf{WS}}$ cannot p-simulate regular resolution. In particular, consider the family of $GT$ formulas, defined by Bonet and Galesi [1999], which separate regular resolution from tree resolution.

*Definition* 4.24. For any $n$ the $GT_n$ formula includes all clauses of $GOP(K_n)$ where $K_n$ is the complete graph on $V = \{1, ..., n\}$ together with *totality* clauses $(x_{i,j} \vee x_{j,i})$ for each $i \neq j$.

As shown by Bonet and Galesi [1999], like the formulas $Peb_{G,S,T}$, these formulas have polynomial-size regular resolution refutations but require exponential-size tree resolution refutations.

Write $G \dashv_{WS} H$ iff $H$ follows from $G$ solely via Weakening and Subsumption. We observe the following simple properties of $\dashv_{WS}$.

PROPOSITION 4.25.

(a) $\dashv_{WS}$ *is transitive. That is, if* $F \dashv_{WS} G$ *and* $G \dashv_{WS} H$ *then* $F \dashv_{WS} H$.
(b) *If* $F \dashv_{WS} H$ *and* $G \dashv_{WS} H$ *then* $F \wedge G \dashv_{WS} H$ .
(c) *For any literal* $x$, *if* $G \dashv_{WS} H$ *then* $G|_x \dashv_{WS} H|_x$.

PROOF. Parts (a) and (b) follow immediately from the definition. Suppose that $x$ is a literal and $G \dashv_{WS} H$. If $C \in G|_x$ then neither $x$ nor $\overline{x}$ appears in $C$ and either $C$ or $(C \vee \overline{x})$ appears in $G$. If $C \in G$ then there is some $D \in H$ with $D \subseteq C$ $D \in H|_x$. If $(C \vee \overline{x}) \in G$ then there is some $D \in H$ with $D \subseteq (C \vee \overline{x})$ and thus $D|_x \subseteq C$ and $D|_x \in H|_x$. Thus (c) follows. □

Let $unitprop(H)$ be the formula obtained from $H$ after applying unit propagations to $H$.

LEMMA 4.26. *If* $G \dashv_{WS} H$ *then there is a restriction* $\pi$ *such that* $G|_\pi \dashv_{WS}$ $unitprop(H)$ *and* $G|_\pi$ *has no unit clauses.*

PROOF. Assume that $\Lambda \notin unitprop(H)$ for otherwise the lemma follows immediately with $G|_\pi = unitprop(G)$. Otherwise let $\pi$ be the set of assignments that are made during unit propagation on $H$. By Proposition 4.25 we have $G|_\pi \dashv_{WS} unitprop(H)$. If $x$ is a unit clause in $G|_\pi$ then, since $\Lambda \notin unitprop(H)$, $unitprop(H)$ must contain $x$ as a unit clause, which is a contradiction. □

We will be interested in formulas $G = GT_n|_\sigma$ and $H = GT_n|_\tau$ such that $G \dashv_{WS} H$. Using Lemma 4.26 we will only need to study this when $G$ and $H$ have no unit clauses and $H$ does not contain the empty clause.

Observe that if $G = GT_n|_\sigma$ has no unit clauses and does not contain the empty clause then $\sigma$ must be transitively closed and so we can identify $\sigma$ with a partial order $<_\sigma$ on $V$.

Given a partial order $<_\sigma$ on $V$ define

—$\sigma^i = \{j \in V \mid i <_\sigma j\}$,
—$minimal(\sigma) = \{i \in V \mid \nexists j \in V, j <_\sigma i\}$,
—$tops(\sigma) = \{k \in V \mid \forall i \in minimal(\sigma), i <_\sigma k\}$, and
—$prune(\sigma)$ to be $<_\sigma$ restricted to $V - tops(\sigma)$.

LEMMA 4.27. *If $G = GT_n|_\sigma \dashv_{WS} H = GT_n|_\tau$ and $G$ and $H$ do not contain $\Lambda$ or any unit clause then $prune(\sigma) = prune(\tau)$.*

PROOF. For any pair $j, k \in V$, if $j$ and $k$ are incomparable in $<_\sigma$ then $G$ contains the clause $(x_{j,k} \bigvee x_{k,j})$ which must also appear in $H$ since $H$ does not contain $\Lambda$ or a unit clause. Therefore $j$ and $k$ are incomparable in $<_\tau$.

Since $G$ does not contain $\Lambda$ or a unit clause, $G$ contains a nonminimality clause $C_i = \bigvee_{j \in V - \sigma^i} x_{j,i}$ of size at least 2 for each $i \in minimal(\sigma)$. Therefore $H$ must contain a clause $D_i \subseteq C_i$ with at least two positive literals whose last coordinate is $i$. This can only be the nonminimality clause $D_i = \bigvee_{j \in V - \tau^i} x_{j,i}$ and thus $i \in minimal(\tau)$ and $\sigma^i \subseteq \tau^i$. Since any $j \notin \sigma^i$ is incomparable to $i$ in $<_\sigma$, it must be incomparable to $i$ in $<_\tau$ so $j \notin \tau^i$. Therefore $minimal(\sigma) = minimal(\tau)$ and each such minimal element has $\sigma^i = \tau^i$. Furthermore by definition $tops(\sigma) = tops(\tau)$.

If $j <_\sigma k$ and $j, k \in V - tops(\sigma)$ then there is some $i \in minimal(\sigma)$ such that $i \not<_\sigma k$. Therefore $i$ is incomparable to both $j$ and $k$ in $<_\sigma$. Therefore $G$ will contain two clauses of size 2 that are the restrictions of the transitivity clauses for the triple $(i, j, k)$, namely $(\neg x_{i,j} \vee x_{i,k})$ and $(\neg x_{k,i} \vee x_{j,i})$. These clauses must also appear in $H$ and the only possible sources for them are the same transitivity clauses in $GT_n$. Therefore $j <_\tau k$.

Therefore for all $j, k \in V - tops(\sigma) = V - tops(\tau)$, $j <_\sigma k$ if and only if $j <_\tau k$ and thus $prune(\sigma) = prune(\tau)$. □

THEOREM 4.28. *Any $\mathsf{FC}^{\mathsf{WS}}$ refutation of $GT_n$ requires at least $2^{n-2}$ nodes.*

PROOF. We show that there are at least $2^{n-2}$ distinct residual formulas in any such refutation, with the property that no two of them can be inferred using Weakening and Subsumption from the same residual subformula.

For any restriction $\rho$ such that $GT_n|_\rho$ does not infer $\Lambda$ via unit propagation, the transitive closure, $\rho^*$, of the relation defined by $\rho$ forms a partial order $<_{\rho^*}$. Call a branch point in an $\mathsf{FC}^{\mathsf{WS}}$ execution *novel* if (1) the residual formula $GT_n|_\rho$ at the branch point does not infer $\Lambda$ by unit propagation and (2) it branches on a variable $x_{i,j}$ such that $i$ and $j$ are in different connected components of the Hasse diagram associated with $<_{\rho^*}$. Observe that if only $n-2$ novel branch points have been made on a path then $tops(\rho^*) = \emptyset$. Furthermore, every consistent branch can be extended until it contains at least $n-2$ novel branch points and the restrictions $\rho$ defining these branches are inconsistent with each other. Therefore there are at least $2^{n-2}$ of them at the novelty level $n-2$ and their transitive closures $\sigma$ all have $prune(\sigma) = <_\sigma$ and disagree about the relative order of some pair of elements.

Let $H = GT_n|_\rho$ be the residual formula at a novel branch point and assume that $G = GT_n|_\pi$ infers $H$ using Weakening and Subsumption. Applying

Lemma 4.26 we obtain a formula $G' = G|_{\pi'}$ for the restriction $\pi'$ such that $unitprop(H) = H|_{\pi'}$, $G' \dashv_{WS} unitprop(H)$, and $G'$ does not have an empty or unit clause (because the branch point associated with $H$ is novel and $G$ infers $H$.) Let $\sigma$ be the restriction that is the combination of $\pi$ and $\pi'$ and let $\tau$ be the restriction that is the combination of $\rho$ and $\pi'$. By construction $\sigma$ and $\tau$ correspond to partial orders on $\{1, \ldots, n\}$. By Lemma 4.27 we must have $<_\tau = prune(\tau) = prune(\sigma) = <_\sigma$.

Now if $G$ is added to $L$ before $H$ in the execution of $\mathsf{FC}^{\mathsf{WS}}$ then there are two cases. Either $G$ is in the subtree below $H$, or $G$ is in a subtree that was traversed before the node corresponding to $H$ is traversed. In this latter case, $G$ is in a subtree to the left of the node corresponding to $H$, and thus there is some branching node where a variable $x_{i,j}$ is queried, on which $\pi$ and $\rho$ disagree. If the latter were to occur, the corresponding extended restrictions $\sigma$ and $\tau$ would retain this disagreement, and $<_\sigma$ and $<_\tau$ would disagree about the relative order of $i$ and $j$. This would contradict the requirement that $<_\tau = <_\sigma$. Therefore any such clause $G$ would have to be in the subtree below $H$. Since these subtrees are disjoint for every pair $H$ and $H'$ of our set of clauses at novelty level $n - 2$, the theorem follows. □

COROLLARY 4.29. $\mathsf{FC}^{\mathsf{WS}}$ *does not polynomially simulate regular resolution.*

Thus, even the strongest of the basic formula caching systems is not strong enough to efficiently simulate regular resolution. In fact, these systems cannot efficiently simulate the *ordered* regular resolution method defined in the original paper of Davis and Putnam [1960] since, as shown by Bonet and Galesi [1999], the formulas $GT_n$ are provable in ordered regular resolution.

However, as we saw earlier, if we augment formula caching by having it return the reason for unsatisfiability as well as the mere fact of unsatisfiability, to obtain $\mathsf{FC}^{\mathsf{W}}_{reason}$ then we can not only efficiently simulate regular resolution but also can efficiently refute formulas that require exponential size $\mathsf{Res}(k)$ refutations.

## 5. CONTRADICTION CACHING SYSTEMS WITH 0-1 SUBSTITUTION

In a preliminary version of this work [Beame et al. 2003], we defined an additional rule for the CC and FC systems that was called the restriction rule. To be consistent with earlier terminology, we should have called this rule the 0-1 substitution rule.

*Definition* 5.1. The *restriction rule* (also called the *0-1 substitution* rule) in a CC system allows $\varphi|_x$ (or $\varphi|_{\overline{x}}$) to be inferred from $\varphi$.

We define the letter "R" to denote the 0-1 substitution (or restriction) rule. Thus CC+WSR is the contradiction caching system that includes weakening, subsumption, and restriction.

We mistakenly claimed that CC+WSR was no more powerful than depth-2 Frege systems. In fact, we show shortly that the restriction rule is surprisingly powerful; in fact adding it makes the system p-equivalent to Extended Frege systems. The proof of this follows the paper of Buss [1995] where he shows

that a standard Frege system plus 0-1 substitution is p-equivalent to Extended Frege. In Pitassi and Urquhart [1995], Buss's result was further refined to show that even a depth-2 Frege system plus 0-1 substitution is p-equivalent to Extended Frege.

*Definition* 5.2 [PITASSI AND URQUHART 1995]. $\mathcal{S}_{0,1}\mathcal{F}_2$ is a refutation system for CNF formulas. It contains the axiom and rules of $\mathcal{F}_2$ plus the 0-1 substitution rule.

The following two lemmas show that CC+WSR is p-equivalent to Extended Frege.

LEMMA 5.3 [PITASSI AND URQUHART 1995]. *$\mathcal{S}_{0,1}\mathcal{F}_2$ is p-equivalent to Extended Frege.*

LEMMA 5.4. CC+WSR *is p-equivalent to* $\mathcal{S}_{0,1}\mathcal{F}_2$.

PROOF. We first show that CC+WSR can p-simulate $\mathcal{S}_{0,1}\mathcal{F}_2$. The axiom, $(x \wedge \overline{x})$ can be derived from the axiom of CC plus an application of Branching. Rule R0' is implicit in the CC system, because CNF formulas are viewed as sets of clauses in the CC systems. Rule R1' is equivalent to Weakening; Rule R2' is equivalent to Subsumption; and clearly the substitution rules are equivalent. It is left to show how to simulate rules R3' and R4'.

First, we show how to simulate rule R4'. Given $(F \wedge x)$, and $(G \wedge \overline{x})$ we want to derive $(F \wedge G)$. First, apply Weakening to obtain $(F \wedge G \wedge x)$ and $(F \wedge G \wedge \overline{x})$. Apply the Restriction rule to obtain $(F \wedge G)|_x$ from $(F \wedge G \wedge x)$. Similarly apply the Restriction rule to obtain $(F \wedge G)|_{\overline{x}}$ from $(F \wedge G \wedge \overline{x})$. Finally apply Branching to obtain $(F \wedge G)$ as desired.

Lastly, rule R3' can be simulated as follows. Given $A \wedge F$ and $B \wedge F$, we want to derive $(A \vee B) \wedge F$. If $A \vee B$ contains some literal $x$ and $B$ contains $\overline{x}$ then we apply Subsumption to derive $x \wedge F$ and $\overline{x} \wedge F$ and then rules R4' followed by R0' (already simulated) to derive $F$, and finally Weakening to derive $(A \vee B) \wedge F$. Similarly, if either $A$ or $B$ contains both $x$ and $\overline{x}$ then we can derive $F$ by Restriction on $x$ and $\overline{x}$ and Branching to return $F$ and then Weakening as before. Otherwise, suppose that $B = (l_1 \vee l_2 \vee \ldots \vee l_k \vee B')$ where $B'$ consists of the literals of $B$ that are contained in $A$. We need to derive $(A \vee l_1 \vee \ldots \vee l_k) \wedge F$.

For each $i = 1, \ldots, k$, apply Weakening to derive

$$l_i \wedge F \qquad (a_i)$$

from $B \wedge F$. Next, apply Restriction and Branching to derive

$$(A \vee l_1) \wedge (A \vee \overline{l}_1) \wedge F \qquad (b)$$

from $(A \wedge F)$. (Restriction is only required if $F$ contains instances of $l_1$ or $\overline{l}_1$; in this case we apply Restriction under $l_1$ and $\overline{l}_1$, which yields $A \vee F|_{l_1}$ and $A \vee F|_{\overline{l}_1}$ since $A$ does not contain $l_1$ or $\overline{l}_1$, and then Branching in order to derive (b).) Next, we apply Subsumption to derive

$$(\overline{l}_1) \wedge (A \vee l_1) \wedge F \qquad (b_1)$$

from (b). Similarly, apply Restriction and Branching followed by Subsumption to derive

$$(\bar{l}_1) \wedge (\bar{l}_2) \wedge (A \vee l_1 \vee l_2) \wedge F \qquad (b_2)$$

from ($b_1$). Continue in this way (applying Restriction and Branching followed by Subsumption) to derive

$$(\bar{l}_1) \wedge \cdots \wedge (\bar{l}_k) \wedge (A \vee l_1 \vee l_2 \vee \cdots \vee l_k) \wedge F. \qquad (b_k)$$

Now by repeated application of rule R4' (which we have already shown how to simulate), we can eventually derive $(A \vee l_1 \vee l_2 \vee \cdots \vee l_k) \wedge F$ from ($b_k$) and ($a_i$), $i = 1, \cdots, k$, as desired.

In the other direction, we want to show how $\mathcal{S}_{0,1}\mathcal{F}_2$ can p-simulate CC+WSR. We have already proven that $\mathcal{F}_2$ can p-simulate CC+WS. From this it is easy to see that $\mathcal{S}_{0,1}\mathcal{F}_2$ can p-simulate CC+WSR, because the substitution rules are equivalent. □

## 6. CONCLUSIONS AND OPEN PROBLEMS

In this article, we have initiated a study of the proof complexity of DPLL proofs augmented with various forms of formula caching. As we have discovered, the complexity of DPLL with caching is surprisingly subtle and counterintuitive. Naively, we expected that adding caching capabilities to tree-like resolution would give us the power of general resolution. However, this intuition is very wrong. In fact, we were unable to come up with any natural and efficiently implementable version of caching that could p-simulate resolution. On the other hand, we were able to define a simple and theoretically implementable version of DPLL with formula caching ($\mathsf{FC}^{\mathsf{W}}_{reason}$), that is sometimes exponentially more powerful than both resolution and Res($k$)!

Can these advantages be translated to practice? One of the most interesting of our new systems is the system $\mathsf{FC}^{\mathsf{W}}_{reason}$. While, in theory, this system can be exponentially more powerful than clause learning, it is not clear if this superiority can be achieved in practice. A challenging task would be to develop an implementation of a variant of $\mathsf{FC}^{\mathsf{W}}_{reason}$ that would be competitive against state-of-the-art complete satisfiability solvers. An important issue is cache lookup efficiency. Successful implementations of clause learning are able to utilize extremely fast lookups using hashing. It remains to be seen whether $\mathsf{FC}^{\mathsf{W}}_{reason}$ lookups can be performed efficiently enough to be of practical value. Some combination of clause learning and formula caching seems to have the best potential of outperforming clause learning by itself. In such an implementation, the partial truth assignment (the "reason") would be stored in the cache. Then a cache lookup could be of one of several types. A cheap lookup would involve checking if the current partial assignment $\rho$ contains $\sigma$, for some $\sigma$ in the cache. This corresponds to a clause learning lookup. A more costly lookup would check if $f$ restricted by $\rho$ is subsumed by $f$ restricted by $\sigma$, for some $\sigma$ in the cache. This corresponds to a formula cache lookup. A reasonable approach would be to perform the cheap lookup at first, and only later in the search attempt to perform the more costly formula cache lookups.

A number of other open problems remain. Most notable are the connections to resolution. What is the weakest, if any, of these caching systems that can p-simulate resolution? It would be particularly interesting if this could be shown for the $\mathsf{FC}^{\mathsf{W}}_{reason}$ proof system. In the reverse direction, can any of the basic FC, $\mathsf{FC}^{\mathsf{W}}$, or $\mathsf{FC}^{\mathsf{WS}}$ systems be p-simulated by resolution or even regular resolution? Secondly, what is the relationship between clause learning and formula caching? Because any clause learning proof is a resolution proof, $\mathsf{FC}^{\mathsf{W}}_{reason}$ can be much more powerful than clause learning. But what about the reverse direction? Does $\mathsf{FC}^{\mathsf{W}}_{reason}$ p-simulate clause learning? Thirdly, is CC+WS p-equivalent to $\mathcal{F}_2$? We were able to show that $\mathcal{F}_2$ p-simulates CC+WS, but what about the reverse direction? In particular, can R4' be simulated by CC+WS? Finally, how do these systems compare with the backtracking models introduced by Alekhnovich et al. [2005] for solving satisfiability?

ACKNOWLEDGMENTS

We would like to thank A. A. Razborov for bringing the errors in the conference version of this article and the likely greater power of the Restriction rule to our attention and for other helpful comments.

REFERENCES

ACHLIOPTAS, D., BEAME, P., AND MOLLOY, M. 2004. A sharp threshold in proof complexity. *J. Comput. Syst. Sci. 68*, 2, 238–268.

ALEKHNOVICH, M., BORODIN, A., BURESH-OPPENHEIM, J., IMPAGLIAZZO, R., MAGEN, A., AND PITASSI, T. 2005. Towards a model for backtracking and dynamic programming. In *Proceedings of the 20th Annual IEEE Conference on Computational Complexity*. 308–322.

ALEKHNOVICH, M., JOHANNSEN, J., PITASSI, T., AND URQUHART, A. 2001. An exponential separation between regular and general resolution. Tech. rep. TR01-56, Electronic Colloquium in Computation Complexity. http://www.eccc.uni-trier.de/eccc/.

BACCHUS, F., DALMAO, S., AND PITASSI, T. 2003a. Algorithms and complexity results for #SAT and Bayesian inference. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science*. IEEE, 340–351.

BACCHUS, F., DALMAO, S., AND PITASSI, T. 2003b. Value elimination: Bayesian inference via backtracking search. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI'03)*. 20–28.

BEAME, P., IMPAGLIAZZO, R., KRAJÍČEK, J., PITASSI, T., PUDLÁK, P., AND WOODS, A. 1992. Exponential lower bounds for the pigeonhole principle. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*. 200–220.

BEAME, P., IMPAGLIAZZO, R., PITASSI, T., AND SEGERLIND, N. 2003. Memoization and DPLL: Formula caching proof systems. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity*. 225–236.

BEAME, P., KAUTZ, H., AND SABHARWAL, A. 2004. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. 22*, 319–351.

BEN-SASSON, E., IMPAGLIAZZO, R., AND WIGDERSON, A. 2000. Near-optimal separation of treelike and general resolution. Tech. rep. TR00-005, Electronic Colloquium in Computation Complexity. http://www.eccc.uni-trier.de/eccc/.

BEN-SASSON, E. AND WIGDERSON, A. 2001. Short proofs are narrow – Resolution made simple. *J. ACM 48*, 2, 149–169.

BONET, M. L., ESTEBAN, J. L., GALESI, N., AND JOHANSEN, J. 2000. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM J. Comput. 30*, 5, 1462–1484.

BONET, M. L. AND GALESI, N. 1999. A study of proof search algorithms for resolution and polynomial calculus. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE, 422–432.

BUSS, S. 1995. Some remarks on the lengths of propositional proofs. *Arc. Math. Logic 34*, 377–394.

BUSS, S. R. 1987. Polynomial size proofs of the pigeonhole principle. *J. Symb. Logic 57*, 916–927.

CHVÁTAL, V. AND SZEMERÉDI, E. 1988. Many hard examples for resolution. *J. ACM 35*, 4, 759–768.

COOK, S. A. AND RECKHOW, R. A. 1977. The relative efficiency of propositional proof systems. *J. Symb. Logic 44*, 1, 36–50.

DAVIS, M. AND PUTNAM, H. 1960. A computing procedure for quantification theory. *Comm. ACM 7*, 201–215.

HAKEN, A. 1985. The intractability of resolution. *Theor. Comput. Sci. 39*, 297–305.

MAJERCIK, S. M. AND LITTMAN, M. L. 1998. Using caching to solve larger probabilistic planning problems. In *Proceedings of the 15th National Conference on Artificial Intelligence*. The AAAI Press/The MIT Press, 954–959.

MARQUES-SILVA, J. P. AND SAKALLAH, K. A. 1996. Grasp – A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*. ACM/IEEE, 220–227.

MONIEN, B. AND SPECKENMEYER, E. 1985. Solving satisfiability in less than $2^n$ steps. *Discr. Appl. Math. 10*, 3, 287–295.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*. ACM/IEEE, 530–535.

PITASSI, T. AND URQUHART, A. 1995. The complexity of the Hajós calculus. *SIAM J. Discr. Math. 8*, 3, 464–483.

ROBSON, J. M. 1986. Algorithms for maximum independent sets. *J. Algor. 7*, 3, 425–440.

SEGERLIND, N., BUSS, S., AND IMPAGLIAZZO, R. 2002. A switching lemma for small restrictions and lower bounds for $k$-DNF resolution. In *Proceedings of the 43rd Annual Symposium on Foundations of Computer Science*. IEEE, 604–613.

URQUHART, A. 1995. The complexity of propositional proofs. *Bull. Symb. Logic 1*, 4, 425–467.

ZHANG, H. 1997. Sato: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*. Lecture Notes in Artificial Intelligence vol. 1249. Springer, 272–275.

ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. H., AND MALIK, S. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*. ACM/IEEE, 279–285.

Received July 2008; revised December 2009; accepted December 2009