Efficient Implementation of Large-Scale Watchlists

Constantin Ruhdorfer and Stephan Schulz

DHBW Stuttgart
mail@ruhdorfer.me schulz@eprover.org

Abstract

In this work, we explore techniques for improving the performance of the automated theorem proving system E when dealing with large *watchlists*. A watchlist can focus the proof search towards so-called *hints*, likely useful intermediate results provided externally. Recently, hints have been automatically extracted from previous proofs, creating massive watchlists and thus making evaluation of new clauses against the wachtlist a performance bottleneck. We introduce a new index for the frequent special case of unit clause hints, taking advantage of the fact that subsumption can be implemented much more efficiently for unit clauses than for the general case. We implement several strategies for exploiting the structure and properties of equational unit clauses. Additionally, we have added a new *soft subsumption* mechanism to E that can abstract away differences of constant or Skolem symbols, effectively allowing a less precise match when evaluating a given clause against the watchlist. We have tested the new mechanisms on a large set of problems taken from the Mizar 40 project, using a large watchlist containing over 300 000 clauses. We show that the usage of the unit clause index significantly increases performance with this given watchlist. The use of soft subsumption shows more mixed results. We believe that most watchlists can take advantage of these techniques and have made them available to the user via E's command line interface.

1 Introduction

Automated theorem provers (ATPs or ATP systems) are programs that accept a set of axioms and a conjecture in a suitable logic, and then try to automatically derive a proof of the conjecture. Many of the most successful theorem provers are based on first-order logic (with equality), an expressive logic with unambiguous semantics for which relatively mature calculi exist. First order logic is semi-decidable. In theory, proofs for valid conjectures can always be found, but proof search for an invalid conjecture may not terminate. This means that an ATP has to search for proofs in an infinite and highly branching search space. Thus, guiding this search is of critical importance for the success of the system. For systems based on forward deduction, the critical choice is which of the many possible intermediate steps should be taken next, i.e. which new formulas should be deduced. This is usually based on simple syntactic criteria (as e.g. described in [11]). However, these heuristics are often insufficient to find complex proofs.

One way to improve the proof search is via *hints*. Originally [13], hints are possible intermediate lemmas provided by the user of the prover. If the prover finds such a lemma (or a more general one), it can focus its search on this lemma. User hints can come from the user's domain expertise and intuition, or possibly from simplified settings. In recent years, we have utilized the same mechanism with a very different source of hints, namely intermediate results contributing to proofs of other theorems in the domain [3]. The system can iteratively build a database of results that are often useful in the domain. In contrast to manually provided hints, the number of hints mined from existing proofs can often be extremely large, and hence evaluating new formulas against the hint set can become quite expensive.

In this work, we are interested in two aspects of the field at hand: Firstly, we want to improve the efficiency of using large sets of hints in guiding a theorem prover (more concretely,

the equational theorem prover **E** [7, 10]). Secondly, we explore a variety of notions about what it means for a hint to match a new formula (or more specifically, *clause*), with the aim of broadening the potential influence of a hint to also influence the selection of clauses that are similar to the hint, not only those that are strictly more general.

2 Preliminiaries

First-order logic with equality We will assume two disjoint sets F and V where F is the set of function symbols and V the set of variable symbols. Function symbols have an associated arity which we will denote with $f/_n$ for symbol f and arity $n \in \mathbb{N}$. Constants are function symbols with $n = 0$.

We will typically use a, b, c to denote constants, f, g, h to denote function symbols and either x, y, z or $\mathrm{X1}, \ldots, \mathrm{X}N$ to denote variables.

The set of syntactically correct terms is denoted by $\mathit{Term}(F, V)$, where $\mathit{Term}(F, V)$ is the smallest set that satisfies the following conditions:

1. $X \in \mathit{Term}(F, V)$ for all $X \in V$
2. $f/_n \in F$, $s_1, \ldots, s_n \in \mathit{Term}(F, V)$ implies $f(s_1, \ldots, s_n) \in \mathit{Term}(F, V)$

An (equational) *atom* is an unordered pair of terms, written as $s \simeq t$. Observe that we handle the non-equational case as a special case where we encode non-equational atoms as equalities with the reserved constant $\$\mathit{true}$, e.g. $p(a) \simeq \$\mathit{true}$. We will typically write non-equational literals in the conventional manner for convenience (e.g. $p(a)$). A *literal* is either an atom, or a negated atom. We write a negative literal as $s \not\simeq t$ and define a negation operator on literals as $\overline{s \simeq t} = s \not\simeq t$ and $\overline{s \not\simeq t} = s \simeq t$. We use $s \mathrel{\tilde{\simeq}} t$ if we do not want to specify the polarity of a literal, or, in a less precise way, let $l, l_1, l_2, \ldots$ stand for arbitrary literals. In this notation $\simeq$ is commutative.

A *clause* is a multiset of literals $\{l_1, l_2, ..., l_n\}$, usually written and always interpreted as a disjunction $l_1 \vee l_2 \vee ... \vee l_n$. A *unit* clause is a clause containing only one literal. We denote the set of all clauses as $\mathit{Clauses}(F, V)$ and the empty clause as $\Box$.

A *substitution* is a mapping $\sigma : V \to \mathit{Term}(F, V)$ with the property that $\mathit{Dom}(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$ is finite. This mapping can be extended to terms, atoms, literals and clauses in the obvious way. If σ is a substitution, we call $\sigma(t)$, $\sigma(l)$, $\sigma(C)$ *instances* of t, l, or C.

Similarly, a *match* from a term (atom, literal, clause) s to another t is a substitution σ such that $\sigma(s) \equiv t$, where $\equiv$ is the syntactic identity.

In most theorem provers for classical first-order logic, proofs are found via contradiction. In other words proof search tries to establish if a given set of clauses is *unsatisfiable*. For generating calculi, new clauses are deduced via a set of inference rules that take one or more (most often two) clauses as premises, and generate a new clause entailed by these premises. If this process eventually derives the empty clause, unsatisfiability has been established (the empty clause is inherently unsatisfiable, and so is any set of clauses that entails it).

Subsumption is a syntactic relation between two clauses. A clause C *subsumes* another clause D, if one of its instances is a multi-subset of the the other, i.e. if $\sigma(C) \subseteq D$. A subsuming clause is more general than the subsumed clause, i.e. the subsumed clause is entailed by the subsuming clause (but not, in general, the other way round). Subsumption plays a double role in this work. On the one hand, in most calculi we can ignore subsumed clauses, and subsumption (the removal of subsumed clauses from the proof search) is a major and important optimisation technique. On the other hand, if a clause subsumed another, it is considered "at least as good" as the first one. The original notion of a clause *matching* a hint is based on subsumption. We

do not require a clause to be identical to a hint to prefer it, but we also prefer clauses that subsume a hint (but note that we further generalise this relation later).

Positions in a term A *potential position* $p \in \mathbb{N}^*$ in a term is defined as a sequence over natural numbers. The empty position is denoted with the special symbol ϵ.

The set of positions in a term t is denoted with $\text{pos}(t)$ and defined recursively by case distinction: If t only consists of a variable symbol $v \in V$ then $\text{pos}(t) = \epsilon$. If on the other hand $t \equiv f(t_1, ..., t_n)$ then $\text{pos}(t) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n,\ p \in \text{pos}(t_i)\}$ with $n \in \mathbb{N}^*$. A position $p \in \text{pos}(t)$ of a term t can be used to refer to the subterm of t at p. To be more exact: if $p = \epsilon$, then $t|_p = t$. Otherwise, $p \equiv i.p'$ and $t \equiv f(t_1, \ldots, t_n)$. In that case, $t_p = t_i|_{p'}$. The top symbol of $x \in V$ is $\text{top}(x) = x$ and the top symbol of $f(t_1, \ldots, t_n)$ is $\text{top}(f(t_1, \ldots, t_n)) = f$.

2.1 Proof search

E is a saturating theorem prover based on the superposition calculus [1]. To prove a conjecture, axioms and the negated conjecture are converted to clause normal form, resulting in a set of clauses that is unsatisfiable if and only if the conjecture holds. The proof state thus is a set of clauses, and the proof search is realised by *saturating* this set of clauses by adding logical consequences that can be deduced from existing clauses by application of a number of inference rules. If this process generates the empty clause as an explicit witness of unsatisfiability, the proof has been concluded.

In practice, this proof search is realised via the *given-clause* algorithm. The proof state is represented by two disjoint sets of clauses, the set U of unprocessed clauses, and the set P of processed clauses. The algorithm repeatedly picks a clause g from U, computes all possible consequences between this given clause and all clauses in P, and adds them to U. It then adds g (the *given clause*) to P. This maintains the invariant that all direct consequences of clauses in P have been computed. In addition to these *generating inferences*, the algorithm can use *simplifying rules* to replace clauses by simpler clauses, or to delete redundant clauses.

The most critical choice point for the given-clause algorithm is the selection of the given clause for each iteration of the main loop. This is traditionally controlled by heuristic evaluations, based on symbol counting (smaller clauses are preferred), clause age (older clauses are preferred), and various combinations and refinements of these measures (compare [12]).

2.2 Watchlist

Large parts of this work focus around the *watchlist* technique which was originally developed by Robert Veroff who named it the *hint strategy* [13]. The strategy was developed for guiding ATP programs in their proof search by comparing newly generated clauses against a list of hints. Such a list of hints is user-provided and usually contains lemmas, facts or otherwise clauses the user suspects might be relevant to the given problem. This technique was first implemented into Otter [5].

In the E ATP system the watchlist mechanism is implemented two-fold as a *dynamic* and a *static* variant [2]. Regardless of the variant used the list is loaded on start-up and stored as a Clause-Set where it is simplified like processed clauses. A Clause-Set is a internal data structure in E that stores clauses using a doubly linked list and provides access to its members via various indices. Every newly generated or processed clause is compared against the watchlist by checking whether or not the new clause matches one or more clauses in the watchlist. If it does it is prioritized for processing.

2.3 Indexing techniques

One of the most important factors when it comes to the performance of ATP systems is efficient indexing. Indexing helps to avoid, or at least reduce, time spent on sequential search within large sets of clauses or terms. E has included several different indices for a while, including *(perfect) discrimination tree indexing* [6], *feature vector indexing* [9] and *fingerprint indexing* [8].

E has been using feature vector indexing for non-unit subsumption [9], and indexes only the processed set of clauses P. Feature vector indexing is particularly suitable for indexing relatively large multi-literal clauses, since it handles the complexity of equation- and literal permutation by using features that are invariant under these permutations. This is a major advantage compared to other approaches of handling subsumption via indices. It does not, however, come into play for relatively small unit clauses, for which better indices exist. One of which is fingerprint indexing which is a technique that samples positions in terms for its indexing representation. We can adjust these sampled positions in a way that takes advantage of the fact that every unit clause exactly consist out two terms (more on that later). First, we will introduce fingerprint indexing for this purpose.

Fingerprint Indexing A *fingerprint index* [8] is as trie over *fingerprints* fp of terms. The *general fingerprint feature function* $\text{gfpf} : Term(F, V) \times \mathbb{N}^* \to F'$ where $F' = F \uplus \{\mathbf{A}, \mathbf{B}, \mathbf{N}\}$ is defined by case distinction:

$$\text{gfpf}(t, p) = \begin{cases} \mathbf{A} & \text{if } p \in \text{pos}(t),\ t|_p \in V \\ \text{top}(t|_p) & \text{if } p \in \text{pos}(t),\ t|_p \notin V \\ \mathbf{B} & \text{if } p = q.r,\ q \in \text{pos}(t) \text{ and } t|_q \in V \text{ for some } q \\ \mathbf{N} & \text{otherwise} \end{cases}$$

Here $\text{top}(x)$ is x if $x \in V$ and f if $x \equiv f(t_1, ..., t_n)$. Given that the *fingerprint feature function* is a function $\text{fpf} : Term(F, V) \to F'$ and is defined by $\text{fpf}(t) = \text{gfpf}(t, p)$ for a fixed $p \in \mathbb{N}^*$. Lastly the *fingerprint function* is defined by $\text{fp} : Term(F, V) \to (F')^n$ for a fixed $n \in \mathbb{N}$. A fingerprint is a vector of elements of F' and is calculated by $\text{fp}(t)$ for a given term t.

For an arbitrary fpf and two terms s and t assume two values $u = \text{fpf}(s)$ and $v = \text{fpf}(t)$. An overview for the compatibility of unification and matching from s onto t, given u and v, is presented in Figure 1.

Unification	f_1	f_2	**A**	**B**	**N**
f_1	**Y**	**N**	**Y**	**Y**	**N**
f_2	**N**	**Y**	**Y**	**Y**	**N**
A	**Y**	**Y**	**Y**	**Y**	**N**
B	**Y**	**Y**	**Y**	**Y**	**Y**
N	**N**	**N**	**N**	**Y**	**Y**

Matching	f_1	f_2	**A**	**B**	**N**
f_1	**Y**	**N**	**N**	**N**	**N**
f_2	**N**	**Y**	**N**	**N**	**N**
A	**Y**	**Y**	**Y**	**N**	**N**
B	**Y**	**Y**	**Y**	**Y**	**Y**
N	**N**	**N**	**N**	**N**	**Y**

Figure 1: Compatibility for unification and matching where f_1 and f_2 are arbitrary but distinct. Taken from [8].

3 Implementation

When we originally introduced the watchlist feature, we expected to work with fairly small watchlists, and decided to use feature vector indexing for all hint matching. However, watchlists now contain several hundred thousand clauses, and evaluating new clauses against the watchlist has become a major bottleneck. To reduce this bottleneck, we have split the watchlist index into a pair of unit and non-unit clause indices, to decrease access times by storing fewer clauses in either by using the most appropriate indexing technique for either set. For a similar idea see [3].

Unit clause index We have implemented a new unit clause index in E based on fingerprint indexing. Since this is a technique for term and not clause indexing we exploit the structure of unit clauses to generate an indexing representation for the given unit clause. We use the fact that all unit clauses are of the form $\{\text{lterm} \doteq \text{rterm}\}$ to construct a new term, represented by a $\$EQN$ cell of the form $\doteq(\text{lterm}, \text{rterm})$, over which we calculate the fingerprint. We do that by alternating between lterm and rterm for sampling positions. Since all indexed terms start with one kind of equality symbol (e.g. $\simeq$ or $\not\simeq$) we can skip it when constructing the fingerprint for a term. The ϵ position is therefore never sampled.

Clauses that are not orientable are inserted twice into the index since changing the orientation of a clause also changes its fingerprint. In the worst-case this would lead to the size of index doubling. We therefore checked old runs of E and found that around 15% of clauses were not orientable and would therefore be inserted twice. Although this is a considerate increase we guess that this would have a negligible impact on performance while designing this data structure. Inserted clauses are simply stored as a pointer in the leaf of the fingerprint trie using a splay tree.

We present an example index in Figure 2 which assumes an example fingerprint function $FPW4$ that samples at $(1, 2, 1.1, 2.1)$ and $F = \{f/_2, g/_1, a/_0, b/_0, c/_0\}$.

We have implemented several fingerprint functions to cover a wide variety of needs. We started with functions that assume full equality in the terms they sample which means that they sample both sides equally: NoIdx (no unit clause index), FPW2, FPW4, FPW6, FPW8 and FPW10 (see Table 1 for details). Although E is an equality based ATP system not all problems are purely equational or equational at all. The same also applies to watchlists. E already categorizes problems based on whether they are non-equational N, somewhat equational S and purely equational P. We use the same mechanism to classify the degree of equality in the given watchlists.

Based on that we alter the strategy used to sample the positions. This is since with increasingly less equational watchlist the right side of a term is more likely to simply be $\$true$ and there is no useful information to be sampled. To address this we also introduced a left only (marked with "L", e.g. FPW2L) and a left leaning (marked with "LL", e.g. FPW2LL) version of each fingerprint function. The left only version will skip position ϵ and continue to only sample positions on the left side, e.g. FPW2L samples at $1, 1.1$ and FPW6L at $1, 1.1, 1.1, 1.1.1, 1.2.1, 1.1.2$. The left leaning version will sample roughly between 2/3 and 3/4 of the positions from the left side, depending on the size of the fingerprint function. Since FPW2 samples so few positions FPW2LL is the same function as FPW2L. For an overview over all strategies please consult Table 1.

While the left leaning version surely is more useful for somewhat equational clause sets, using the strategy will still result in many sampled positions that are non-existent and therefore not useful when it comes to matching. We therefore propose yet another strategy to be used for

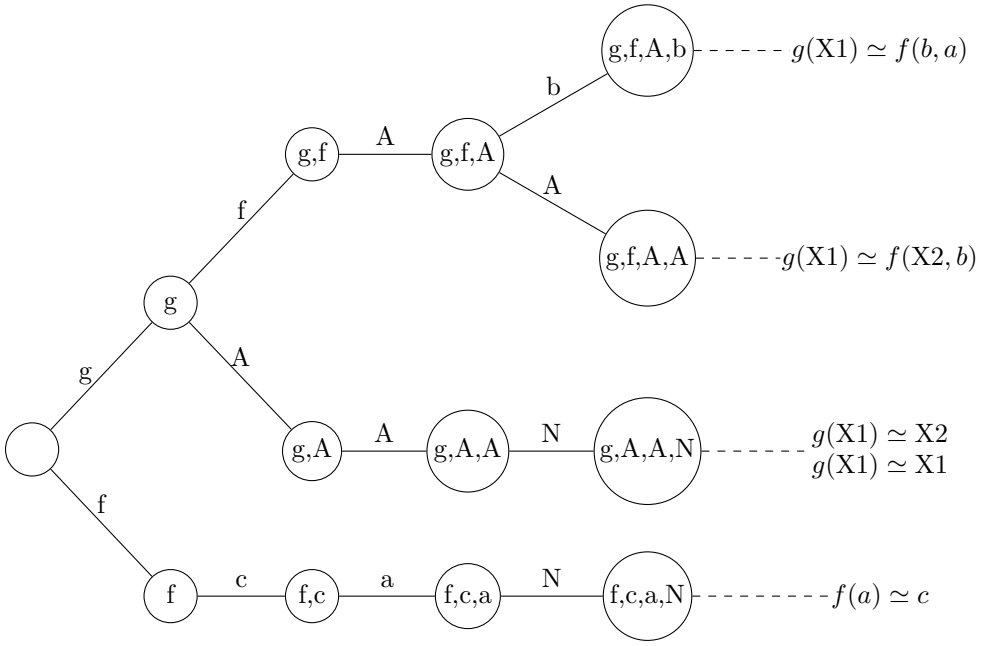


Figure 2: Example unit clause index given FPW4

partly equational clause sets which we will denote by "*Flex" (e.g. FPW2Flex). A flex type strategy is one where we first classify the input based on whether or not the right side of the term is $\$true$. We then use an L type sampling method on the term if it is or a balanced one if it is not. On the one hand this allows us to better exploit the structure of the given term while on the other hand this will result in the index returning some terms that are not actually a match if they had been sampled with the same fingerprint function. This is since now their fingerprints might be sampled from different positions. While that seems troublesome at first this is not really an issue for two reasons: (i) It is unlikely that this will affect many terms since one unmatchable symbol in the fingerprint will already reject the match and (ii) since fingerprint indexing is an non-perfect indexing method to begin with we need to check whether the given clause subsumes the every returned clause anyway. In the worst case we will need to check slightly more results for subsumption.

We implemented all these options into E and made them available through Es domain-specific language (DSL). On top of that we also implemented an automatic mode (available as "auto") that maps a N watchlist to an "LL" type function, an S watchlist to an "L" type function and a P watchlist to a normal strategy. As a basis for that we used "FPW6" since we expect it to perform well across many different watchlists.

Clause abstraction We have also implemented a clause abstraction mechanism in E for the watchlist feature. The mechanism supports two modes of operation: One abstracting constants and one abstracting skolem symbols. If turned on our implementation will rewrite all clauses that are inserted into or checked against the watchlist to adhere to the abstraction. This effectively allows for less precise matches against the watchlist.

Strategy name	Positions sampled
NoIdx	-
FPW2	1, 2
FPW2L	1, 1.1
FPW2Flex	FPW2 or FPW2L (see text)
FPW4	1, 2, 1.1, 2.1
FPW4L	1, 1.1, 1.2, 1.1.1
FPW4LL	1, 2, 1.1, 1.2,
FPW6	1, 2, 1.1, 2.1, 1.1.1, 2.1.1
FPW6L	1, 1.1, 1.1, 1.1.1, 1.2.1, 1.1.2
FPW6LL	1, 2, 1.1, 2.1, 1.1.1, 1.2.1
FPW6Flex	FPW6 or FPW6L (see text)
FPW8	1, 2, 1.1, 1.2, 2.1, 2.2, 1.1.1, 2.1.1
FPW8L	1, 1.1, 1.2, 1.1.1, 1.2.1, 1.1.2, 1.1.1.1, 1.1.1.2
FPW8LL	1, 2, 1.1, 2.1, 1.1.1, 1.2.1, 1.1.2, 2.1.1
FPW10	1, 2, 1.1, 1.2, 2.1, 2.2, 1.1.1, 1.1.2, 2.1.1, 2.1.2
FPW10L	1, 1.1, 1.2, 1.1.1, 1.2.1, 1.1.2, 1.1.1.1, 1.2.1.1, 1.1.2.1, 1.1.1.2
FPW10LL	1, 2, 1.1, 2.1, 1.1.1, 1.2.1, 1.1.2, 2.1.1, 1.1.1.1, 2.1.1.1

Table 1: Sampling strategies overview

If constants are to be abstracted all constants are rewritten to the first constant met during the proof for an untyped problem and for a typed one to the first constant met with the appropriate sort. That is given a clause $c_1 = \{f(b,c) \dot{\simeq} g(X1), g(X1) \dot{\simeq} g(a)\}$, $F = \{f/_2, g/_1, a/_0, b/_0, c/_0\}$ and the first met constant a we will rewrite the clause c_1 to $c_1' = \{f(a,a) \dot{\simeq} g(X1), g(X1) \dot{\simeq} g(a)\}$ assuming a, b, c share the same sort.

The mechanism works similarly for abstracting skolem symbols where we rewrite them to the first met skolem symbol with the same type. We have also made this an available option to turn on through Es DSL.

4 Experimental Results

We tested on Intel Xeon E5-2698 v3 CPUs at 2.30 GHz using the Linux 3.19.0-25-generic kernel in 64-bit mode. All tests were run with a time limit of 720 seconds and a limit of 10.000 generated clauses. For orchestrating the experiments we used the ATPy library[1].

[1] Written by Jan Jakubův; Online accessible at https://github.com/ai4reason/pyprove

4.1 Unit clause index

For testing we used a strategy[2] provided by the automated reasoning group at Czech Technical University in Prague who also provided a watchlist based on previous runs of the system. The watchlist contained 367.408 clauses of which 153.997 are unit. The strategy was run against a tenth of the Mizar 40 project [4] amounting to 5787 problems. We have made all of this data and the E version used available at http://eprover.eu/E-eu/SoftWatch.html.

		All runs		Successfull runs[3]	
Strategy	# proofs	Total time	Mean time	Total time	Mean time
NoIdx (baseline)	1679	462880.8	79.9	59131.6	35.2
FPW2	1685	397159.8	68.6	49357.8	29.2
FPW2L	1694	346104.6	59.7	37460.5	22.1
FPW2Flex	1686	405478.3	70.0	50220.2	29.7
FPW4	1684	402009.4	69.4	49601.9	29.4
FPW4L	1688	413876.4	71.5	51165.3	30.3
FPW4LL	1688	416879.8	72.0	51639.3	30.5
FPW6	1685	397808.6	68.7	49398.9	29.3
FPW6L	1688	407316.0	70.3	50617.3	29.9
FPW6LL	1685	408941.3	70.6	50594.4	30.0
FPW6Flex	1689	407502.0	70.4	50673.4	30.0
FPW8	1686	399834.2	69.1	49678.4	29.4
FPW8L	1689	407277.4	70.3	50638.9	29.9
FPW8LL	1686	408242.1	70.5	50574.7	29.9
FPW10	1685	400546.6	69.2	49778.3	29.5
FPW10L	1689	409653.3	70.7	51000.8	30.1
FPW10LL	1686	409560.7	70.7	50808.7	30.1

Table 2: Performance of various indices (in seconds).

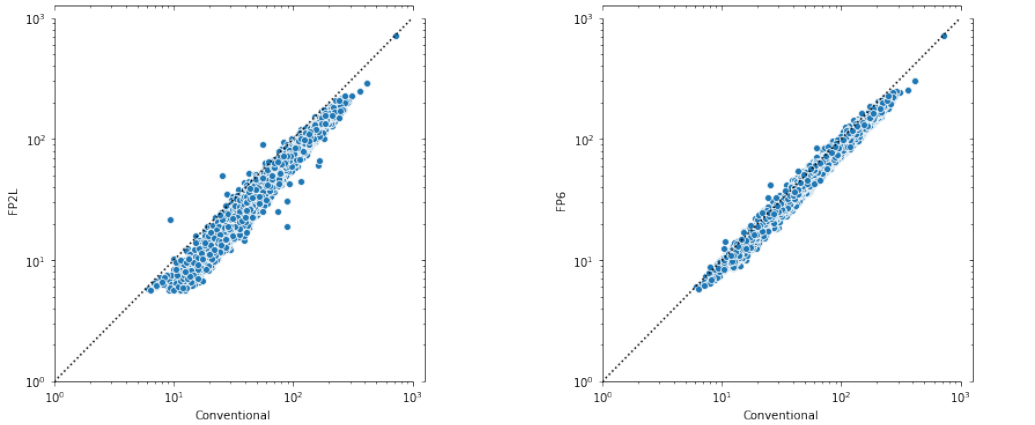
Table 2 shows different versions of the index and their performance. Observe that we compare our implementation against a version of E that only uses feature vector indexing as an indexing technique for the watchlist. This version is refereed to as “Conventional” or “NoIdx” (no index) since it is missing the unit clause index. To measure performance we observe the runtime E given the set of problems. We chose to compare runtimes since the proof search for any given problem nearly always stays the same between different indexing strategies. To verify that the runs indeed stay the same we compared a random subsample of proof searches. In our comparisons we will differentiate between the runtime for all problems and only those that were deemed successful [3].

[2]The exact options given were:
−−definitional−cnf=24 −−split−aggressive −−simul−paramod −−forward−context−sr −−destructive−er−aggressive −−destructive−er −−prefer−initial−clauses −tKBO6 −winvfreqrank −c1 −Ginvfreq −F1 −−delete−bad−limit=150000000 −WSelectMaxLComplexAvoidPosPred −H’(1∗ConjectureTermPrefixWeight(PreferProcessed,1,3,0.1,5,0,0.1,1,4),1∗ConjectureTermPrefixWeight(PreferWatchlist,1,3,0.5,100,0,0.2,0.2,4),1∗Refinedweight(PreferWatchlist,4,300,4,4,0.7),1∗RelevanceLevelWeight2(PreferWatchlist,0,1,2,1,1,1,200,200,2.5,9999.9,9999.9),1∗StaggeredWeight(PreferWatchlist,1),1∗SymbolTypeweight(PreferWatchlist,18,7,−2,5,9999.9,2,1.5),2∗Clauseweight(PreferWatchlist,20,9999,4),2∗ConjectureSymbolWeight(PreferWatchlist,9999,20,50,−1,50,3,3,0.5),2∗StaggeredWeight(PreferWatchlist,2))’ −−free−numbers

[3]Runs with exit status ”Theorem” or ”CounterSatisfiable”

We expected to find similar results compared with the original fingerprint paper [8]. Meaning that we expected to find that a fingerprint size of 6 is a good balance of trie depth and clause distribution. Although comparing FPW6 with the no indexed version yields an improvement of 16.35% for all runs and 19.65% for all runs that were successful we find that other strategies were even more successful. Surprisingly FPW2L yielded the best performance performing 28.86% for all runs and 44.87% for all successful runs.

While this index increases performance on average Figure 3 shows that the actual performance is dependent on the problem itself. Please note the figures' logarithmic scale. Notice that most strategies perform very similar. This very likely is an effect of testing on the same watchlist where all strategies perform well, but one can exploit some inherent structure of the watchlist better. The "auto" strategy is not listed since for the watchlist tested it would evaluate to the performance of FPW6LL.



(a) FPW2L runtime comparison (in seconds). (b) FPW6 runtime comparison (in seconds).

Figure 3: Conventional vs FPW6 vs FPW2L

Lastly, we were interested in examining wether the different strategies solve the same problems or instead if they are proving different problems. We found that they overwhelmingly do. That is to say the intersection of problems solved by all strategies, which is a very limiting factor, is 1668 problems big (includes the baseline). Given that most strategies solve around 1680 problems this means that the overlap of solved problems is 99%. We compiled the runtimes on these 1668 problems in Table 3.

The problems solved by all strategies are very likely the easier problems in the whole set. We therefore might expect the performance timings to be better than the the runtimes observed in Table 2 but this is not the case.

4.2 Validating the use of fingerprint indexing

One central claim of this paper is that exploiting the structure of unit clauses leads to better performance compared to a standard feature vector index. This claim can easily be verified by comparing the performance of E when either indexing technique is only filled with unit clauses. This can be achieved by using a watchlist that only contains unit clauses. We do that

Index	Total time	Mean time
NoIdx (baseline)	58499.3	35.1
FPW2	48716.8	29.1
FPW2L	36409.3	21.8
FPW4	49043.4	29.3
FPW4L	50299.4	30.1
FPW4LL	50757.9	30.4
FPW6	48756.3	29.2
FPW6L	49765.5	29.8
FPW6LL	49932.8	29.9
FPW8	48968.1	29.3
FPW8L	49727.5	29.7
FPW8LL	49824.9	29.8
FPW10	49061.0	29.3
FPW10L	50070.1	30.0
FPW10LL	50044.4	29.9

Table 3: Performance of various indices on problems every strategy solved (in seconds).

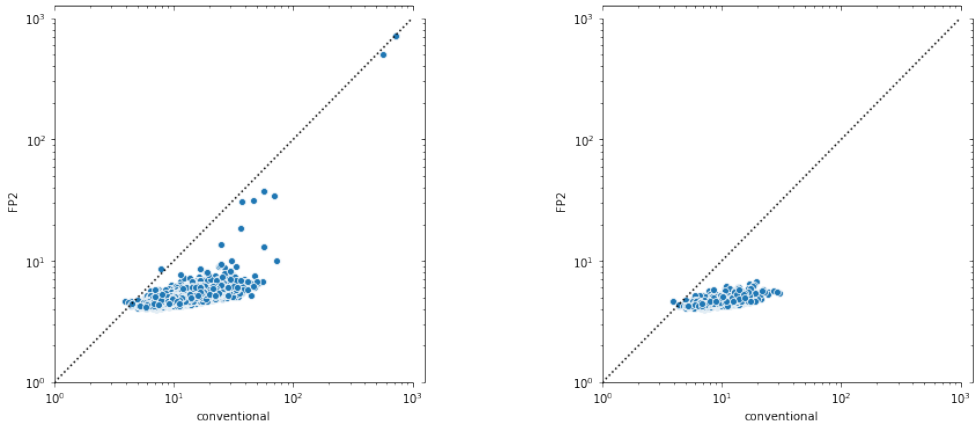
by removing all non-unit clauses from the watchlist described above. We did not alter the set of problems.

		All runs		Successfull runs[3]	
Index	# proofs	Total time	Mean time	Total time	Mean time
FVI (baseline)	1409	80189.6	13.8	13935.6	9.8
FPW2	1412	31460.9	5.4	6560.4	4.6
FPW2L	1410	43707.7	7.6	9434.5	6.7
FPW4	1415	45400.1	7.8	10006.0	7.0
FPW4L	1413	42495.9	7.3	9466.7	6.7
FPW4LL	1409	45186.3	7.8	9937.3	7.0
FPW6	1413	46181.0	7.9	10159.6	7.1
FPW6L	1415	41666.4	6.3	8928.4	7.2
FPW6LL	1414	38294.2	6.6	8182.9	5.8
FPW8	1412	46142.1	7.9	10188.5	7.2
FPW8L	1416	43763.7	7.6	9622.5	6.8
FPW8LL	1410	44122.2	7.6	9695.4	6.8
FPW10	1416	46537.6	8.0	10262.2	7.2
FPW10L	1410	45082.0	7.8	9967.4	7.1
FPW10LL	1409	43593.1	7.5	9666.9	6.9

Table 4: Performance of various indices on a unit clause only watchlist (in seconds).

Table 4 shows that a fingerprint index using the FPW2 fpf significantly outperforms all other used indices but especially standard feature vector indexing. This does indeed verify that exploiting the structure of unit clauses for sampling yields better performance. Notice that the fastest strategy FPW2 does not solve the most problems. This is most likely due to random

variations in the proof search.



(a) All runs (in seconds). (b) Successful runs (in seconds).

Figure 4: FPW2 vs standard Feature Vector Indexing using only unit clauses.

Figure 4 shows that this is true across all problems when comparing runtimes. Interestingly a trie depth of only two outperforms all other strategies tried which goes against the original papers finding where a balance of trie depth and leaf size performs best [8].

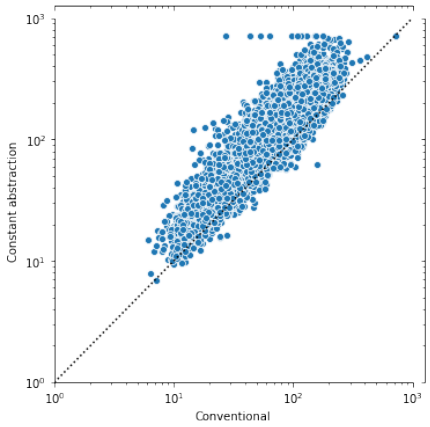
4.3 Clause Abstraction

We used a similar test setup for determining performance of the clause abstraction feature. We have used the previous options for the prover (as stated in footnote 2) together with either adding the flag −−watchlist−clause−abstraction=constant or −−watchlist−clause−abstraction =skolem respectively. The results are shown in Table 5.

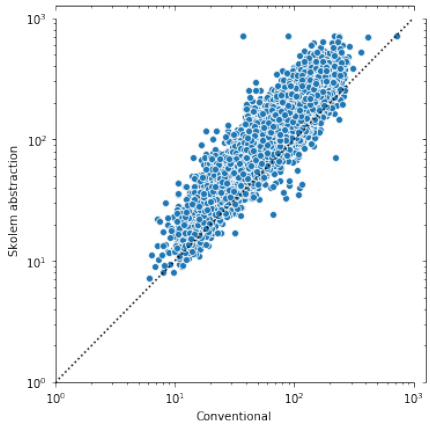
		All runs		Successfull runs[3]	
Strategy	# proofs	Total time	Mean time	Total time	Mean time
No	1679	462880.8	79.9	59131.6	35.2
Constant	1612	796494.2	137.6	96920.8	60.1
Skolem	1612	770005.3	133.0	96920.8	60.1

Table 5: Performance of various clause abstraction strategies (in seconds).

We find that the performance varies from problem to problem. This is especially true when abstracting constant symbols where some problems started to run out of time. Compare this to just one problem for all other strategies tested (see Figure 3). This effect is clearly visible in the scatter plot Figure 5a.



(a) Constant abstraction (in seconds).



(b) Skolem symbol abstraction (in seconds).

Figure 5: Conventional vs clause abstraction

5 Future Work

We have identified at least two more interesting areas of study. Firstly instead of rewriting a skolem symbol to one of the same arity, we would also be interested in rewriting complete skolem terms to a constant. Secondly, we are also interested in generalizing the idea of splitting the watchlist indices into even more smaller ones to increase performance.

6 Conclusion

In this work, we have presented a special unit clause index for the watchlist feature based on fingerprint indexing. We explored the performance for several strategies with that index given a large watchlist of 300 000 clauses and showed that the index largely increases performance compared to a version without the index. We conclude that the performance of the index is dependent on the watchlist, its structure and the strategy used. We believe that most watchlists can benefit from this index.

We have also introduced some mechanism to E that allow for less precise matches on the watchlist. While that showed more mixed results in terms of performance it is an interesting topic that would benefit from additional exploration.

Acknowledgements: Special thanks to the Automated Reasoning Group at Czech Technical University in Prague for providing the watchlist, the problem files and the experimental environment.

References

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

[2] Zarathustra Goertzel, Jan Jakubův, Stephan Schulz, and Josef Urban. ProofWatch: Watchlist guidance for large theories in E. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving: 9th International Conference, Oxford, UK*, pages 270–288. Springer, 2018.

[3] Zarathustra Goertzel, Jan Jakubuv, and Josef Urban. Enigmawatch: Proofwatch meets ENIGMA. *CoRR*, abs/1905.09565, 2019.

[4] Cezary Kaliszyk and Josef Urban. Mizar 40 for mizar 40. *CoRR*, abs/1310.2805, 2013.

[5] William McCune. Otter 2.0. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 663–664, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[6] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.*, 9(2):147–167, October 1992.

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

[8] Stephan Schulz. Fingerprint Indexing for Paramodulation and Rewriting. In Bernhard Gramlich, Ulrike Sattler, and Dale Miller, editors, *Proc. of the 6th IJCAR, Manchester*, volume 7364 of *LNAI*, pages 477–483. Springer, 2012.

[9] Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *LNAI*, pages 45–67. Springer, 2013.

[10] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pacal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.

[11] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In Nicola Olivetti and Ashish Tiwari, editors, *Proc. of the 8th IJCAR, Coimbra*, volume 9706 of *LNAI*, pages 330–345. Springer, 2016.

[12] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, page 330–345, Berlin, Heidelberg, 2016. Springer-Verlag.

[13] Robert Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *Journal of Automated Reasoning*, 16(3):223–239, Jun 1996.