Efficient E-matching for SMT Solvers

Leonardo de Moura and Nikolaj Bjørner

Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
{leonardo, nbjorner}@microsoft.com

Abstract. Satisfiability Modulo Theories (SMT) solvers have proven highly scalable, efficient and suitable for integrating theory reasoning. However, for numerous applications from program analysis and verification, the ground fragment is insufficient, as proof obligations often include quantifiers. A well known approach for quantifier reasoning uses a matching algorithm that works against an E-graph to instantiate quantified variables. This paper introduces algorithms that identify matches on E-graphs incrementally and efficiently. In particular, we introduce an index that works on E-graphs, called *E-matching code trees* that combine features of substitution and code trees, used in saturation based theorem provers. E-matching code trees allow performing matching against several patterns simultaneously. The code trees are combined with an additional index, called the *inverted path index*, which filters E-graph terms that may potentially match patterns when the E-graph is updated. Experimental results show substantial performance improvements over existing state-of-the-art SMT solvers.

1 Introduction

SMT solvers based on a DPLL(T) [1] framework have proven highly scalable, efficient and suitable for integrating theory reasoning. However, for numerous applications from program analysis and verification, an integration of decision procedures for the ground fragment is insufficient, as proof obligations often include quantifiers for capturing frame conditions over loops, summarizing auxiliary invariants over heaps, and for supplying axioms of theories that are not already equipped with ground decision procedures. A well known approach for incorporating quantifier reasoning with ground decision procedures is used in the Simplify theorem prover [2]. Simplify uses an E-matching algorithm that works against an E-graph to instantiate quantified variables, where the E-matching problem is defined as:
input: A set of ground equations E, a ground term t and a term p possibly containing variables.
output: The set of substitutions θ, modulo E, over the variables in p, such that $E \models t \simeq \theta(p)$. Two substitutions are equivalent if their right hand sides are pairwise congruent modulo E.

The E-graph, which maintains congruence relations, is modified during a backtracking search. Each modification to the E-graph may enable new instantiations. E-matching is also used in the several other state-of-the-art SMT solvers:

CVC3 [3], Fx7 [4], Verifun [5], Yices [6], Zap [7]. The Stanford Pascal Verifier [8] already included patterns for generating ground instances of axioms. These approaches are also tightly coupled with software verification applications, as found in for instance ESC/Java [9] and Boogie [10, 11].

This paper introduces algorithms that identify matches on E-graphs efficiently and incrementally. In particular, we introduce an index that works on E-graphs, called *E-matching code trees* that combines features of substitution and code trees, used in saturation based theorem provers. E-matching code trees allow performing matching against several patterns simultaneously. The code trees are combined with an additional index, called the *inverted path index*, which filters E-graph terms that may potentially match patterns after modifications in the E-graph. The choice and design of these indices reflect upon measured runtime overheads. While E-matching is in theory NP-hard [12], and the number of matches can be exponential in the size of the E-graph, the practical overhead of using E-matching for quantifier instantiation turns out to be searching and maintaining sets of patterns that can efficiently retrieve new matches as soon as E-graph operations introduce them.

Quantifier reasoning is native to saturation based theorem provers where resolution and superposition are the main mechanisms for producing inferences. However, few implementations and experiments have been reported in these systems for reasoning in the context of theories, despite long running attention [13]. Theory resolution [14] provides a framework for adding theory reasoning (as for instance, unification modulo associativity and commutativity) to such systems. In practice, some decision procedures are included in SNARK, including Allen's Interval Temporal Logic and theories used in the Amphion system [15]. Recently [16] investigated an integration of CVC-lite and SPASS, and combinations with super-position calculi and DPLL and BDDs are investigated in haRVey [17].

2 Background

Let Σ be a *signature* consisting of a set of function symbols, and $\mathcal{V}$ be a set of variables. Each function symbol f is associated with a nonegative integer, called the *arity* of f, denoted $\mathit{arity}(f)$. If $\mathit{arity}(g) = 0$, then g is a constant symbol. The set of terms $T(\Sigma, \mathcal{V})$ is the smallest set containing all constant and variable symbols such that $f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{V})$ whenever $f \in \Sigma$, $\mathit{arity}(f) = n$, and $t_1 \ldots t_n \in T(\Sigma, \mathcal{V})$. A *f-application* is a term of the form $f(t_1, \ldots, t_n)$. The set of ground terms is defined as $T(\Sigma, \emptyset)$. In our context, the set of non ground terms is called *patterns*. We use p, $f(p_1, \ldots, p_n)$, and x, y, z to range over patterns, and t, $f(t_1, \ldots, t_n)$, and a, b, c to range over ground terms.

In our context, a *substitution* is a mapping from variables to ground terms. Given a substitution β, we denote by $\beta(p)$ the ground term obtained by replacing every variable x in the pattern p by $\beta(x)$.

A binary relation R over T is an *equivalence relation* if it is reflexive, symmetric, and transitive. An equivalence relation induces a partition of T into *equivalence classes*. Given a binary relation R, its *equivalence closure* is the smallest

equivalence relation that contains R. A binary relation R on $T(\Sigma, \emptyset)$ is *monotonic* if $\langle f(t_1, \ldots, t_n), f(t'_1, \ldots, t'_n)\rangle \in R$ whenever $f \in \Sigma$ and $\langle t_i, t'_i\rangle \in R$ for all i in $1 \ldots n$. A *congruence relation* is a monotonic equivalence relation. Given a binary relation R on $T(\Sigma, \emptyset)$, its *congruence closure* is the smallest congruence relation that contains R.

An *E-graph* data-structure maintains the *congruence closure* of a binary relation $E = \{(t_1, t'_1), \ldots, (t_k, t'_k)\}$ given incrementally (on-line) as a sequence of operations $\mathit{union}(t_1, t'_1), \ldots, \mathit{union}(t_k, t'_k)$. Each equivalence class is represented by its *representative*. For each term t in the E-graph, $\mathit{find}(t)$ denotes the representative of the equivalence class that contains t, $\mathit{class}(t)$ denotes the equivalence class that contains t, $\mathit{apps}_f(t)$ denotes the set of terms $f(t_1, \ldots, t_n)$ such that $f(t_1, \ldots, t_n) \in \mathit{class}(t)$, $\mathit{apps}(f)$ denotes the set of all f-applications in the E-graph, $\mathit{parents}(t)$ denotes the set of terms $f(\ldots, t', \ldots)$ in the E-graph such that $t' \in \mathit{class}(t)$, $\mathit{parents}_f(t)$ is a subset of $\mathit{parents}(t)$ which contains only f-applications, and $\mathit{parents}_{f.i}(t)$ is a subset of $\mathit{parents}_f(t)$ which contains only f-applications where the *i-th* argument t_i is in $\mathit{class}(t)$. The set $\mathit{ancestors}(t)$ is the smallest set such that $\mathit{parents}(t) \subseteq \mathit{ancestors}(t)$, and $\mathit{ancestors}(t_p) \subseteq \mathit{ancestors}(t)$ whenever $t_p \in \mathit{ancestors}(t)$. We suppress references to E-graphs from the above functions, as there is always only one E-graph during proof search.

2.1 SMT solvers

Modern SMT solvers combine boolean satisfiability solvers based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure, and T-solvers capable of deciding the satisfiability of conjunctions of T-atoms. In this paper, T-atoms are equalities between ground terms, and quantified formulas. A T-solver maintains a state that is an internal representation of the atoms asserted so far. This solver must provide operations for updating the state by asserting new atoms, checking whether the state is consistent, and backtracking. The solver maintains a stack of *checkpoints* that mark consistent states to which the solver can backtrack.

Most SMT solvers incorporate quantifier reasoning using *E-matching*. Semantically, the formula $\forall x_1, \ldots, x_n.F$ is equivalent to the infinite conjunction $\bigwedge_\beta \beta(F)$ where β ranges over all substitutions over the x's. In practice, solvers use heuristics to select from this infinite conjunction those instances that are "relevant" to the conjecture. The key idea is to treat an instance $\beta(F)$ as relevant whenever it contains enough terms that are represented in the current E-graph. That is, non ground terms p from F are selected as *patterns*, and $\beta(F)$ is considered relevant whenever $\beta(p)$ is in the E-graph. An abstract version of the *E-matching* algorithm is shown in Fig. 1. The set of relevant substitutions for a pattern p can be obtained by taking $\bigcup_{t \in E} \mathit{match}(p, t, \emptyset)$. The abstract matching procedure returns all substitutions that E-match a pattern p with term t. That is, if $\beta \in \mathit{match}(p, t, \emptyset)$ then $E \models \beta(p) = t$, and conversely, if $E \models \beta(p) = t$, then there is a β' congruent (when interpreted as a set of equalities) to β such that $\beta' \in \mathit{match}(p, t, \emptyset)$. In [18], this claim is justified in more detail by observing that the abstract matcher may be viewed as a congruence proof search procedure.

$$
\begin{aligned}
match(x,t,S) &= \{\beta \cup \{x \mapsto t\} \mid \beta \in S, x \notin dom(\beta)\} \ \cup \\
&\quad \{\beta \mid \beta \in S, \mathit{find}(\beta(x)) = \mathit{find}(t)\} \\
match(c,t,S) &= S \ \mathit{if}\ c \in class(t) \\
match(c,t,S) &= \emptyset \ \mathit{if}\ c \notin class(t) \\
match(f(p_1,\ldots,p_n),t,S) &= \bigcup_{f(t_1,\ldots,t_n)\in class(t)} match(p_n,t_n,\ldots,match(p_1,t_1,S))
\end{aligned}
$$

Fig. 1. E-matching (abstract) algorithm.

3 E-matching Abstract Machine

It is usual in automated deduction to compile terms into code that can be efficiently executed at retrieval time. The compiler produces code for a real machine, or for a virtual machine as in the case of Prolog's WAM [19]. In this section, we propose an abstract machine for E-matching, its instructions, compilation process, and interpretation. Memory of the abstract machine is divided in the following way:

- register *pc* for storing the current instruction.
- an array of registers *reg*[] for storing ground terms.
- a stack *bstack* for backtracking.

The basic instruction set of our abstract machine consists of: init, bind, check, compare, choose, yield, and backtrack. The semantics of these instructions, shown in Fig. 2, corresponds closely to the steps used by the abstract matching procedure; so if a pattern p is compiled into a code sequence starting with the instruction *instr*, then the set $match(p,t,\emptyset)$ is retrieved by storing t in $reg[0]$, setting *pc* to *instr*, and executing the instruction stored in *pc*. This claim is justified in more detail in [18], by observing, for instance, that the compare instruction handles repeated variable occurrences in a pattern. At the moment choose is not relevant, it will be used when we discuss the case of matching against many patterns simultaneously. The instruction bind creates a backtracking point, the idea is to try all f-applications in the equivalence class of the term stored in $reg[i]$. The effect of the backtrack instruction is to pop the *top* of the backtracking stack, *bstack*, and perform the instruction stored in *top*. The abstract machine terminates when the backtracking stack *bstack* is empty. For convenience, we define the function *cont* on instructions. On all above instructions but yield, *cont* returns *next*; for example, $cont(\mathsf{check}(i,t,next)) = next$. The pattern $f(x_1, g(x_1,a), h(x_2), b)$ can be compiled in the following code sequence:

$$\mathsf{init}(f, \mathsf{check}(4, b, \mathsf{bind}(2, g, 5, \mathsf{compare}(1, 5, \mathsf{check}(6, a, \mathsf{bind}(3, h, 7, \mathsf{yield}(1, 7)))))))$$

In the rest of the paper, we represent code sequences using *labeled instructions*. A labeled instruction will be written as a pair of the form $n : instr$, where n is

$\mathsf{init}(f, next)$	assuming $reg[0] = f(t_1, \ldots, t_n)$ $reg[1] := t_1;\ \ldots;\ reg[n] := t_n$ $pc := next$
$\mathsf{bind}(i, f, o, next)$	$push(bstack,\ \mathsf{choose\text{-}app}(o, next, apps_f(reg[i]), 1))$ $pc :=$ backtrack
$\mathsf{check}(i, t, next)$	**if** $find(reg[i]) = find(t)$ **then** $pc := next$ **else** $pc :=$ backtrack
$\mathsf{compare}(i, j, next)$	**if** $find(reg[i]) = find(reg[j])$ **then** $pc := next$ **else** $pc :=$ backtrack
$\mathsf{choose}(alt, next)$	**if** $alt \neq nil$ **then** $push(bstack,\ alt)$ $pc := next$
$\mathsf{yield}(i_1, \ldots, i_k)$	yield substitution $\{x_1 \mapsto reg[i_1], \ldots, x_k \mapsto reg[i_k]\}$ $pc :=$ backtrack
backtrack	**if** $bstack$ is not empty **then** $pc := pop(bstack)$ **else stop**
$\mathsf{choose\text{-}app}(o, next, s, j)$	**if** $\lvert s\rvert \geq j$ **then** **let** $f(t_1, \ldots, t_n)$ be the j^{th} term in s. $reg[o] := t_1;\ \ldots;\ reg[o + n - 1] := t_n$ $push(bstack,\ \mathsf{choose\text{-}app}(o, next, s, j + 1))$ $pc := next$ **else** $pc :=$ backtrack

Fig. 2. Semantics of abstract machine instructions.

the label/address, and *instr* is the instruction itself. Using labeled instructions, the code sequence above is represented as:

$$\mathsf{init}(f, n_1),\ n_1 : \mathsf{check}(4, b, n_2),\ n_2 : \mathsf{bind}(2, g, 5, n_3),\ n_3 : \mathsf{compare}(1, 5, n_4),$$
$$n_4 : \mathsf{check}(6, a, n_5),\ n_5 : \mathsf{bind}(3, h, 7, n_6),\ n_6 : \mathsf{yield}(1, 7)$$

In the function *compile*(W, V, o), W (*working set*) is a mapping from register indices to patterns, V (*variables*) is mapping from variables to register indices, and o (*offset*) contains the value of the next available register index. The elements of the working set W can be processed in any order, but in our implementation an entry $i \mapsto f(p_1, \ldots, p_n)$ is only processed when W does not contain an entry $i \mapsto t$ or $i \mapsto x_k$. The idea is to give preference to instructions that do not produce backtracking points.

4 E-matching Code Trees

The time spent on matching patterns with shared structures can be minimized by combining different code sequences in a *code tree*. Code trees were introduced in [20] in the context of saturation based theorem provers. They are used for forward subsumption and forward demodulation in the Vampire theorem

$$
\begin{aligned}
\mathit{compile}(f(p_1,\ldots,p_n)) &= \mathsf{init}(f, \mathit{compile}(\{1 \mapsto p_1,\ldots,n \mapsto p_n\}, \emptyset, n+1))\\
\mathit{compile}(\{i \mapsto t\} \cup W, V, o) &= \mathsf{check}(i, t, \mathit{compile}(W, V, o)),\ \text{when } t \text{ is a ground term.}\\
\mathit{compile}(\{i \mapsto x_k\} \cup W, V, o) &= \mathit{compile}(W, V \cup \{x_k \mapsto i\}, o),\ \text{if } x_k \notin \mathit{dom}(V)\\
&= \mathsf{compare}(i, V(x_k), \mathit{compile}(W, V, o)),\ \text{otherwise.}\\
\mathit{compile}(\{i \mapsto f(p_1,\ldots,p_n)\} \cup W, V, o) &= \mathsf{bind}(i, f, o, \mathit{compile}(W', V, o+n)),\\
&\quad \text{where } W' = W \cup \{o \mapsto p_1,\ldots,(o+n-1) \mapsto p_n\}\\
\mathit{compile}(\emptyset, \{x_1 \mapsto i_1,\ldots,x_k \mapsto i_k\}, o) &= \mathsf{yield}(i_1,\ldots,i_k)
\end{aligned}
$$

Fig. 3. Algorithm for compiling patterns into code sequences.

$\mathsf{init}(f, n_1)$
- $n_1 : \mathsf{choose}(n_9, n_2),\ n_2 : \mathsf{bind}(2, g, 3, n_3)$
  - $n_3 : \mathsf{choose}(n_6, n_4),\ n_4 : \mathsf{check}(3, a, n_5),\ n_5 : \mathsf{yield}(1, 4)$
  - $n_6 : \mathsf{choose}(\mathit{nil}, n_7),\ n_7 : \mathsf{compare}(1, 3, n_8),\ n_8 : \mathsf{yield}(1, 4)$
- $n_9 : \mathsf{choose}(\mathit{nil}, n_{10}),\ n_{10} : \mathsf{check}(2, b, n_{11}),\ n_{11} : \mathsf{bind}(1, h, 5, n_{12})$
  - $n_{12} : \mathsf{choose}(n_{14}, n_{13}),\ n_{13} : \mathsf{yield}(5, 6)$
  - $n_{14} : \mathsf{choose}(\mathit{nil}, n_{15}),\ n_{15} : \mathsf{bind}(6, g, 7, n_{16}),\ n_{16} : \mathsf{compare}(5, 7, n_{17}),\ n_{17} : \mathsf{yield}(5, 8)$

Fig. 4. Code tree for $\{f(x, g(a, y)),\ f(x, g(x, y)),\ f(h(x, y), b),\ f(h(x, g(x, y)), b)\}$.

prover [21]. The code trees presented in this section are similar to *substitution trees*[22], also used in saturation based theorem provers. The key advantage of using code and substitution trees is that matching work common to multiple patterns is "factored out." This advantage results in substantial speedups over a naive approach that would repeatedly match a term against each pattern. A code tree for a small set of patterns is shown in Fig. 4. Each line can be viewed as node (or *code block*) in the tree, indentation is used to suggest a parent-child relationship between nodes, the instruction choose is used to create branches/choices in the tree. The node starting at label n_1 (n_9) contains the instruction(s) common for matching the first and second (third and fourth) patterns. In E-matching code trees, the yield instruction must also store the quantifier that should be instantiated with the yielded substitution, this information is suppressed to simplify the exposition. Our code trees are also very similar to context trees [23]. The main differences with other code, substitution, and context trees, include the use of a stack to handle both backtracking and the branching that arize from matching in the context of an E-graph.

In general, to maintain a code tree C for a dynamically changing set of patterns P, one has to implement operations for integrating and removing code from the tree. In our context, patterns are added to the code tree when the DPLL(T) engine asserts an atom that represents a quantified formula, and are

$$
\begin{aligned}
\mathit{insert}(\mathsf{init}(f,n), f(p_1,\ldots,p_m)) &= \mathit{try}(n, \{1 \mapsto p_1, \ldots, m \mapsto p_m\}, \mathit{nreg}(\mathsf{init}(f,n)), [\mathsf{init}(f,n)], [\,]) \\
\mathit{try}(\mathsf{choose}(a,n), W, o, C, I) &= \bot, \text{ if } C = [\,] \\
&= \mathit{seq}(C, \mathit{firstfit}(\mathsf{choose}(a,n), W, o)), \text{ if } I = [\,], \\
&= \mathit{branch}(C, \mathit{seq}(I, \mathsf{choose}(a,n)), W, o), \text{ otherwise.} \\
\mathit{try}(\mathsf{yield}(i_1,\ldots,i_k), W, o, C, I) &= \bot, \text{ if } C = [\,], \\
&= \mathit{branch}(C, \mathit{seq}(I, \mathsf{yield}(i_1,\ldots,i_n)), W, o), \text{ otherwise.} \\
\mathit{try}(\mathit{instr}, W, o, C, I) &= \mathit{try}(\mathit{cont}(\mathit{instr}), W, o, C, I \,\hat{}\, [\mathit{instr}]), \text{ if } \mathit{compatible}(\mathit{instr}, W) = \bot, \\
&= \mathit{try}(\mathit{cont}(\mathit{instr}), \mathit{compatible}(\mathit{instr}, W), C \,\hat{}\, [\mathit{instr}], I), \text{ otherwise.} \\
\mathit{firstfit}(\mathsf{choose}(a,n), W, o) &= \mathsf{choose}(a, \mathit{try}(n, W, o, [\,], [\,])), \text{ if } \mathit{try}(n, W, o, [\,], [\,]) \neq \bot, \\
&= \mathsf{choose}(\mathit{firstfit}(a, W, o), n), \text{ otherwise.} \\
\mathit{firstfit}(\mathit{nil}, W, o) &= \mathsf{choose}(\mathit{nil}, \mathit{compile}(W, \emptyset, o)) \\
\mathit{seq}([\,], \mathit{fchild}) &= \mathit{fchild} \\
\mathit{seq}(\mathsf{check}(i,t,n) : I, \mathit{fchild}) &= \mathsf{check}(i, t, \mathit{seq}(I, \mathit{fchild})) \\
\mathit{seq}(\mathsf{compare}(i,j,n) : I, \mathit{fchild}) &= \mathsf{compare}(i, j, \mathit{seq}(I, \mathit{fchild})) \\
\mathit{seq}(\mathsf{bind}(i,f,o,n) : I, \mathit{fchild}) &= \mathsf{bind}(i, f, o, \mathit{seq}(I, \mathit{fchild})) \\
\mathit{branch}(C, \mathit{fchild}, W, o) &= \mathit{seq}(C, \mathsf{choose}(\mathsf{choose}(\mathit{nil}, \mathit{compile}(W, \emptyset, o)), \mathit{fchild}))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{compatible}(\mathsf{check}(i,t,n), \{i \mapsto t'\} \cup W) &= W, \text{ if } \mathit{find}(t) = \mathit{find}(t') \\
\mathit{compatible}(\mathsf{compare}(i,j,n), \{i \mapsto x, j \mapsto x\} \cup W) &= \{i \mapsto x\} \cup W \\
\mathit{compatible}(\mathsf{bind}(i,f,o,n), \{i \mapsto f(p_1,\ldots,p_m)\} \cup W) &= W \cup \{o \mapsto p_1, \ldots, (o+m-1) \mapsto p_m\} \\
\mathit{compatible}(\mathit{instr}, W) &= \bot, \text{ otherwise.}
\end{aligned}
$$

Fig. 5. Algorithm for insertion into an E-matching code tree.

removed when the DPLL(T) engine backtracks. This usage pattern simplifies the insertion and removal operations. In our implementation, each function symbol is mapped to a unique code tree headed by an init instruction. The algorithm for insertion of new patterns into a code tree is shown in Fig. 5.

Function $\mathit{try}(\mathit{instr}, W, o, C, I)$ traverses a code block accumulating instructions compatible (incompatible) with the working set W in the list C (I), it returns $\bot$ if the code block does not contain any instruction compatible with W. A code block always terminates with a choose or yield instruction. When the code block is fully compatible (i.e., I is empty), the insertion should continue in one of its children. Like substitution trees, there may be several different ways to insert a pattern. The algorithm presented uses a *first fit* (function *firstfit*) strategy when selecting a child block. In our concrete implementation, all children are inspected and the one with the highest number of compatible instructions is used. Function $\mathit{seq}(C, \mathit{fchild})$ returns a code block composed of the instructions in C, whose first child is *fchild*, $\mathit{branch}(C, \mathit{fchild}, W, o)$ returns a code block composed of the instruction in C, and two children: *fchild*, and the code block produced

by the compilation of the working set W. Function $compatible(instr, W)$ returns $\perp$ if the instruction *instr* is not compatible with the working set W, otherwise it returns an updated W by factoring in the effect of *instr*. Function $nreg(c)$ returns the maximum register index used in the code tree c plus one. The yield instruction is always considered incompatible because, as mentioned before, each one is associated with a different quantifier. The init instruction is always compatible because we use a different code tree for each root function symbol. In the context of DPLL(T), removal of code trees follow a chronological backtracking discipline, so it suffices to store old instructions from modified *next* fields in a *trail stack*.

5 Incrementality

The operation $union(t_1, t_2)$ has a potential side-effect of producing new matches. For example, a term $f(a, b)$ matches the pattern $f(g(x), y)$ with a potentially new substitution whenever the operation $union(a, g(c))$ is executed.

The Simplify theorem prover [2, page 409] uses two techniques to identify new terms and patterns that become relevant for matching: *mod-time optimization* and *pattern-element optimization*. *Mod-time optimization* is used to identify relevant terms, and is based on the fact that the operation $union(t_1, t_2)$ may change the set of terms congruent to $t_p \in ancestors(t_1) \cup ancestors(t_2)$. The time needed to traverse the ancestors of a term t can be minimized by marking already visited terms. Marks are removed after every round of matching. When experimenting with this approach we found that most of the ancestors do not produce new matches, and the overhead of traversing them is significant. *Pattern-element optimization* is used to identify relevant patterns. The main idea is to identify when the operation *union* is not relevant for a pattern. A pair of function symbols (f, g) is a *parent-child* pair (*pc-pair*) of a pattern p, if p contains a term of the form:

$$f(\ldots, g(\ldots), \ldots)$$

A pair (not necessarily distinct) of function symbols (f, g) is a *parent-parent* pair (*pp-pair*) of a pattern p, if p contains two distinct occurrences of the variable x of the form:

$$f(\ldots, x, \ldots), \quad g(\ldots, x, \ldots)$$

A $union(t_1, t_2)$ is *pc-relevant* for some *pc-pair* (f, g) of a pattern p whenever

$$(parents_f(t_1) \neq \emptyset \wedge apps_g(t_2) \neq \emptyset) \vee (parents_f(t_2) \neq \emptyset \wedge apps_g(t_1) \neq \emptyset)$$

A $union(t_1, t_2)$ is *pp-relevant* for some *pp-pair* (f, g) of a pattern p whenever

$$(parents_f(t_1) \neq \emptyset \wedge parents_g(t_2) \neq \emptyset) \vee (parents_f(t_2) \neq \emptyset \wedge parents_g(t_1) \neq \emptyset)$$

Assuming that any ground term occurring in a pattern is viewed as a constant symbol, then a $union(t_1, t_2)$ cannot produce new instances for a pattern p if it is not relevant for any *pc-pair* or *pp-pair* of p. The cost of this optimization

is minimized using *approximated sets*, as they are called in [2], these are also known as Bloom filters [24], which are like real sets except that membership and overlap tests may return false positives. Each equivalence class representative t is associated with two approximated sets of function symbols: *funs*(t) and *pfuns*(t), where *funs*(t) is the approximated set of function symbols in *class*(t), and *pfuns*(t) is the approximated set of functions symbols in *parents*(t).

5.1 Inverted path index

Even with mod-time and pattern-element optimizations, many of the matches found are redundant. In this section, we propose a new technique to identify new terms and patterns that become relevant for matching.

An *inverted path string* over a signature Σ is either the empty string Λ, or $f.i.\pi$, where π is an inverted path string, $f \in \Sigma$, and i is an integer. Intuitively, we can view inverted path strings as a child-to-root path. For example, the inverted path string $g.1.f.2$ is a path to term $f(a, g(h(x), c))$ from subterm $h(x)$.

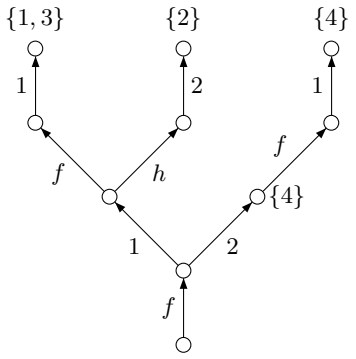


Fig. 6. Inverted path index for *pc-pair* (f, g) and patterns $f(f(g(x), a), x)$, $h(c, f(g(y), x))$, $f(f(g(x), b), y)$, $f(f(a, g(x)), g(y))$.

Given a set of terms T and an inverted path string π, *collect*(π, T) is the set of ancestor terms reached from T following the path π. This set comprises a super-set of terms that participate in new E-matches after a *union* operation. We furthermore seek a sufficiently tight set to avoid redundant E-matching calls.

The function *collect* can be formalized as:

$$\begin{aligned} \mathit{collect}(\Lambda, T) &= T \\ \mathit{collect}(f.i.\pi, T) &= \mathit{collect}(\pi, \{f(t_1, \ldots, t_n) \mid f(t_1, \ldots, t_n) \in \mathit{parents}_{f.i}(t), t \in T\}) \end{aligned}$$

For example, suppose $\mathit{pfuns}(t_1) = \{f\}$, $\mathit{funs}(t_2) = \{g\}$, and $h(x, f(g(y), a))$ is a pattern. Then, $\mathit{collect}(h.2.f.1, \{t_1\})$ contains all terms that may produce new instances for $h(x, f(g(y), a))$ after executing $\mathit{union}(t_1, t_2)$. Collecting the set of potentially useful candidates for matching per pattern is wasteful when a set of patterns share the same *pc/pp-pairs* and furthermore share portions of the inverted paths. We therefore share repeated prefixes from inverted path strings in an *inverted path index*, which has the form of a trie τ. The nodes of τ consist of a list of branches pointing to children together with a set of patterns (corresponding to a code tree) that share the path down to the node. Thus, a node is of the form $\langle [f_1.i_1.\tau_1, \ldots, f_k.i_k.\tau_k], P \rangle$, where τ_j are nodes, $f_j.i_j$ are different

function, integer pairs, and P is a set of patterns. An example of an inverted path index is given in Fig. 6. Adapting a definition of *collect* to inverted path indices is immediate:

$$\mathit{collect}(\langle [f_1.i_1.\tau_1, \ldots f_k.i_k.\tau_k], P\rangle, T) = \{(P,T) \mid P \neq \emptyset\} \cup \bigcup_{j=1}^{k} \mathit{collect}(\tau_j, \{f_j(t_1,\ldots,t_n) \mid f_j(t_1,\ldots,t_n) \in \mathit{parents}_{f_j.i_j}(t), t \in T\})$$

Inverted path indices are particularly useful in situations where one has, for example, different instances of frame axioms using similar patterns: $f(t_1, y, g(z))$, $\ldots, f(t_n, y, g(z))$.

6 Additional instructions

6.1 Multi-patterns

Sometimes it makes sense to instantiate a set of quantified variables only when a set of patterns, called *multi-pattern* is matched. In order to support multi-patterns, a new kind of instruction has been added: continue. The semantics of this instruction is given in Fig. 7. The instruction $\mathsf{continue}(f, o, \mathit{next})$ *chooses* an f-application and updates the registers from o to $o + \mathit{arity}(f) - 1$ with its arguments. For example, the multi-pattern $\langle f(x,a,y), g(z,x)\rangle$ is compiled in the following code sequence:

$$\mathsf{init}(f, n_1),\ n_1 : \mathsf{check}(2, a, n_2),\ n_2 : \mathsf{continue}(g, 4, n_3),\ n_3 : \mathsf{compare}(1, 5, n_4),\ n_4 : \mathsf{yield}(1, 3, 4)$$

In our experiments, we observed that a considerable amount of time was spent matching multi-patterns. The problem is that the instruction $\mathsf{continue}(f, o, \mathit{next})$ is re-executed too many times when the number of f-applications in the E-graph is significant. Considering the code sequence above, a g-application chosen by the continue instruction is only useful to yield an instance if the compare instruction succeeds, that is, the second argument of the chosen g-application is in the same equivalence class of the term stored in register 1. Based on this observation, we added another instruction for compiling multi-patterns: join. The semantics of this instruction is given in Fig. 7. The instruction $\mathsf{join}(i, \pi, o, \mathit{next})$ chooses a candidate from a set of terms reachable from the term stored in register i following the inverted path string π. When a multi-pattern $\langle p_1, \ldots, p_n\rangle$ is compiled, if p_i contains a variable x that also occurs in p_j for $j < i$, then a join can be used instead of a continue instruction, and π is the path from x to p_i. If there is more than one variable, then we select the one with the shallowest path. Using the join instruction the multi-pattern $\langle f(x,a,y), g(z,x)\rangle$ is compiled in the following code sequence:

$$\mathsf{init}(f, n_1),\ n_1 : \mathsf{check}(2, a, n_2),\ n_2 : \mathsf{join}(1, g.2, 4, n_3),\ n_3 : \mathsf{yield}(1, 3, 4)$$

The instruction $\mathsf{compare}(1, 5, n_4)$ is unnecessary, since the join will only select g-applications which the second argument is in the same equivalence class of the term stored in register 1.

$\mathsf{continue}(f, o, next)$	$push(bstack, \mathsf{choose\text{-}app}(o, next, apps(f), 1))$ $pc := \mathsf{backtrack}$
$\mathsf{join}(i, \pi, o, next)$	$push(bstack, \mathsf{choose\text{-}app}(o, next, collect(\pi, \{reg[i]\}), 1))$ $pc := \mathsf{backtrack}$
$\mathsf{filter}(i, fs, next)$	**if** $fs \cap funs(reg[i]) \neq \emptyset$ **then** $pc := next$ **else** $pc := \mathsf{backtrack}$

Fig. 7. Semantics of additional instructions

6.2 Filters

Consider the pattern $f(g(x), h(y))$; it is compiled in the following sequence of instructions:

$$\mathsf{init}(f, 2, n_1),\ n_1 : \mathsf{bind}(1, g, 3, n_2),\ n_2 : \mathsf{bind}(2, h, 4, n_3),\ n_3 : \mathsf{yield}(3, 4)$$

Suppose we are trying to match term $f(a, b)$, and $class(a)$ contains n g-applications, but $class(b)$ does not contain any h-application. In this scenario, a lot of wasteful work is performed when interpreting the instructions above, the second bind will fail n times. We address this problem by introducing a new instruction that performs forward pruning: filter. The semantics of this new instruction is shown in Fig. 7. The idea of the new instruction is to use the approximated set $funs(t)$ to quickly test whether the equivalence class of a term t contains an f-application or not. Using the new instruction, the pattern $f(g(x), h(y))$ is compiled as:

$$\mathsf{init}(f, n_1),\ n_1 : \mathsf{filter}(1, \{g\}, n_2),\ n_2 : \mathsf{filter}(2, \{h\}, n_3),\ n_3 : \mathsf{bind}(1, g, 3, n_4),$$
$$n_4 : \mathsf{bind}(2, h, 4, n_5),\ n_5 : \mathsf{yield}(3, 4)$$

The filter instruction is also used for saving unnecessary backtracking prior to a sequence of choose instructions each followed by a bind to a function in fs.

7 Implementation issues

Relevancy Simplify retains some of the structure of the input formula as an and-or tree. It then implements a tableau style search: to refute a disjunction, each disjunct is refuted independently. Refuting a conjunction only requires retaining each conjunct. In tableau form, the proof rules used by Simplify are:

$$\frac{\bigvee\{\ell_1, \ldots \ell_k\}}{\ell_1 \mid \ldots \mid \ell_k} \qquad \frac{\neg\bigvee\{\ell_1, \ldots, \ell_k\}}{\neg\ell_1, \ldots, \neg\ell_k} \qquad \frac{\neg\neg\ell}{\ell}$$

The tableau search has the side-effect of eliminating irrelevant literals from the scope of a branch. DPLL(T) based solvers do not have this property, as the search assigns a boolean value to potentially all atoms appearing in a goal. For

example, when clausifying $\ell_1 \vee (\ell_2 \wedge \ell_3)$ using a Tseitin [25] style algorithm we obtain the set of clauses:

$$\{\ell_1, \ell_{aux}\},\ \{\ell_{aux}, \neg\ell_2, \neg\ell_3\},\ \{\ell_2, \neg\ell_{aux}\},\ \{\ell_3, \neg\ell_{aux}\}$$

Now, suppose that ℓ_1 is assigned true. In this case, ℓ_2 and ℓ_3 are clearly irrelevant and truth assignments to ℓ_2 and ℓ_3 need not be used, but the Tseitin encoding, which creates a set of clauses, makes the act of discovering this difficult.

The advantage of using relevancy is profound if literals that are pruned from the scope of a branch may produce new quantifier instantiations. We have therefore retained some of the traits of relevancy in our DPLL(T) solver. Our solution does not change how the SAT solver works with respect to case-split heuristics, unit propagation, conflict resolution, etc. Instead, we convert to CNF using a variation of Tseitin algorithm, keep the input formula, and map every (Tseitin) auxiliary variable to a node in the original formula.

Initially, only the auxiliary variable corresponding to the root in the original formula is marked as *relevant*. Relevancy is then propagated to subformulas using the following rules, which effectively simulate the tableau rules. Assume ℓ is marked as relevant. First let ℓ be shorthand for $\bigvee\{\ell_1, \ldots, \ell_k\}$, if ℓ is assigned *true*, then the first child ℓ_i that gets assigned *true* is marked relevant. If ℓ is assigned *false*, then all children are marked relevant. If ℓ is shorthand for $\neg\ell'$, then ℓ' is marked as relevant as well.

Congruent terms If two terms $f(t_1, \ldots, t_n)$ and $f(t'_1, \ldots, t'_n)$ are congruent, then it is wasteful to try to match both of them, since the set of substitutions produced for each of them will be equivalent. Therefore, it suffices to consider only one term from each set of congruent applications for the bind, continue and join instructions, and when considering new candidates for matching.

Eager vs. Lazy instantiation Finding the right instantiations prior to case splits can have the effect of pruning the search space dramatically. On the other hand, eager instantiation of quantifiers that are not helpful in closing branches may amplify the search space. A bi-polar approach to instantiation tactics does not seem to work in general; we found that benchmarks where patterns were supplied by the tools generating the quantified formulas worked best with eager instantation, whereas benchmarks that do not include patterns cannot be solved by eagerly instantiating all quantifiers whenever some subterm can be matched. We therefore collect run-time statistics for when quantifiers are useful for closing branches. Useful quantifiers are promoted to eager instantations, while quantifiers that were not useful are demoted to a lazy instantiation round when other options have been exhausted. The detailed description of the priority queues used for this scheme is elaborated upon in [18].

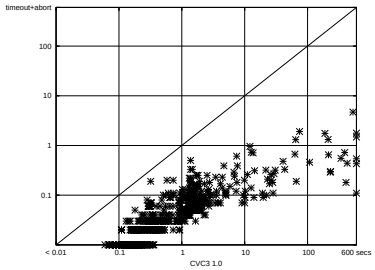
Deleting clauses Quantifier instantiation has a side-effect of producing new clauses containing new atoms into the search space. Retaining these clauses over

backtracking is useless if the new clauses were not helpful in closing the branch. A two-tiered [26] combination of SAT solvers address this problem by using different solvers after (a lazy) quantifier instantiation. Work that was potentially useful for other branches has to be reproduced using other means. In our implementation, we use a single SAT solver, but delete clauses generated from quantifier instantiation when backtracking. Conflict clauses and their literals are on the other hand not deleted.

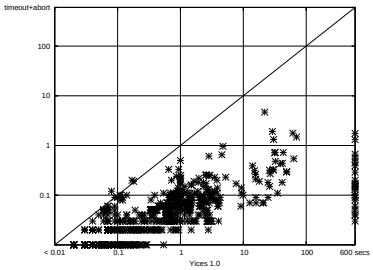
8 Experiments

The experiments were conducted using a 32bit Pentium 4 processor running at 3.6Ghz, 2Gb of memory, and 2Mb of cache. The timeout was set to 10 minutes. We compared our prover, Z3, against CVC3 1.0, Simplify, Yices 1.0, and Zap 2.0. The comparison used more than 3000 publically available benchmarks. It includes the SMT-LIB [27] AUFLIA/simplify, ESC/Java, and Boogie benchmarks.[1] The first set is in SMT-LIB format, and the other two in Simplify format. The most challenging benchmarks from the SMT-LIB AUFLIA benchmarks were derived from the ESC/Java benchmarks. At the time of writing, the SMT-LIB format did not have a standard for specifying patterns for quantified formulas. Most of the benchmarks use linear arithmetic. Fig. 8, 9 and 10 compare Z3 with the other provers, the choice of prover/benchmark set is based on the limitations of the input format accepted by each prover. Each point on the plots represents a benchmark. On each plot the y-axis is the CPU time, in seconds, taken by our prover, and x-axis is for the other prover. Points below the diagonal are then benchmarks where our prover is faster. Points on the rightmost vertical edge are problems where a solver ran out of memory or time. Fig. 11 contains a summary of the experimental results. It also includes a Boogie (non trivial) program verification task: an *s-expression* simplification module which contains 500 lines of code and 32 procedures. The default quantifier instantiation strategy in Z3 uses: code trees, inverted path index, and eager instantiation. The table includes other five different settings for Z3: lazy quantifier instantiation (*lazy*), lazy quantifier instantiation without code trees (*lazy wo. code trees*), eager instantiation without any support for incremental E-matching (*eager wo. inc.*), eager instantiation using the mod-time optimization (*eager mod-time*), eager instantiation using inverted path index but without code tress (*eager wo. code trees*). For each set of benchmarks, the table contains the number of successfully proved instances, and the total time in seconds spent on instances where the solver did not timeout. As can be seen, the Z3 default strategy is very effective. E-matching code trees and the inverted path index are particularly useful in non trivial instances such as the s-expression simplifier.

[1] The benchmarks are also available at http://research.microsoft.com/~leonardo/CADE07

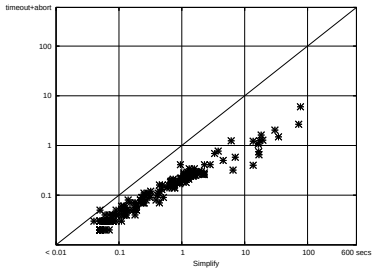


(a) Z3 vs. CVC3 1.0

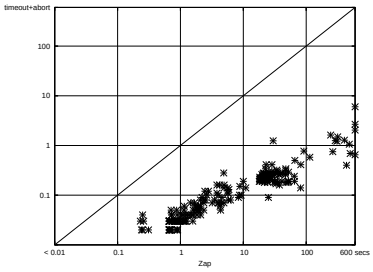


(b) Z3 vs. Yices 1.0

Fig. 8. SMT-LIB Benchmarks



(a) Z3 vs. Simplify

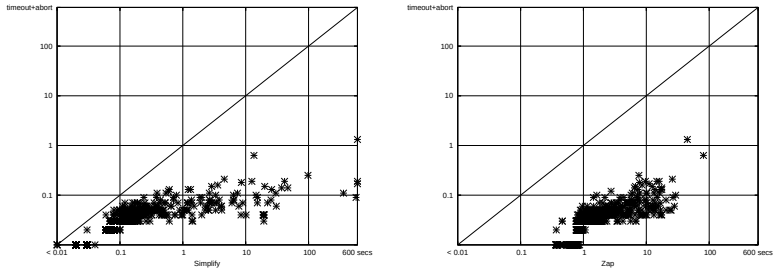


(b) Z3 vs. Zap 2.0

Fig. 9. ESC/Java Benchmarks

9 Conclusion

We have introduced an abstract machine for E-matching. It combines two indices: the *E-matching code trees* which could efficiently handle matching a term against a large set of patterns simultaneously, and *inverted path indexing*, which narrowly and efficiently finds a superset of terms that will match a set of patterns. Other results of the paper are a new approach for handling multi-patterns, and the use of filters inside of an E-matching procedure. Simple and useful heuristics for handling quantifiers in SMT solvers were also presented. Experimental results show that our new solver outperforms the most competitive SMT solvers that support quantifiers. Possible extensions to the approach include using *context trees* [23] for additional sharing, adding instructions to optimize for large alphabets, and extending *inverted path indexing* to a perfect filter for linear patterns.



(a) Z3 vs. Simplify (b) Z3 vs. Zap 2.0

Fig. 10. Boogie Benchmarks

	ESC/Java		Boogie		S-expr Simplifier	
	# valid	time	# valid	time	# valid	time
Simplify	2331	499.03	903	1851.29	18	10985.80
Zap	2222	6297.04	901	2612.64	22	777.78
Z3 (*lazy*)	2331	212.81	907	157.2	32	2904.27
Z3 (*lazy wo. code trees*)	2331	224.14	907	240.44	28	2369.00
Z3 (*eager wo. inc.*)	2331	1495.07	907	229.2	10	2410.52
Z3 (*eager mod-time*)	2331	85.1	907	39.79	32	1341.38
Z3 (*eager wo. code trees*)	2331	48.28	907	26.85	32	654.62
Z3 (*default*)	**2331**	**45.22**	**907**	**18.47**	**32**	**194.54**

Fig. 11. Experimental results: summary.

References

1. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: CAV 04. LNCS 3114 (2004) 175–188
2. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3) (2005) 365–473
3. Barrett, C., Berezin, S.: CVC Lite: A New Implementation of the Cooperating Validity Checker. In: CAV '04. LNCS 3114 (2004)
4. Moskal, M., Lopuszański, J.: Fast quantifier reasoning with lazy proof explication. `http://nemerle.org/~malekith/smt/smt-tr-1.pdf` (2006)
5. Flanagan, C., Joshi, R., Saxe, J.B.: An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Laboratories, Palo Alto (2004)
6. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: CAV'06. LNCS 4144, Springer-Verlag (2006) 81–94
7. Ball, T., Lahiri, S.K., Musuvathi, M.: Zap: Automated theorem proving for software analysis. In: LPAR 2005. LNCS 3835 (2005) 2–22
8. Nelson, G.: Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center (1981)

9. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI. (2002) 234–245
10. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 2005-70, Microsoft Research (2005)
11. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSIS 2004. LNCS 3362, Springer (2005) 49–69
12. Kozen, D.: Complexity of finitely presented algebras. In: STOC. (1977) 164–177
13. Slagle, J.R.: Automatic theorem proving with built-in theories including equality, partial ordering, and sets. J. of the ACM **19(1)** (1972) 120–135
14. Stickel, M.E.: Automated deduction by theory resolution. J. Autom. Reasoning **1**(4) (1985) 333–355
15. Baalen, J.V., Roach, S.: Using decision procedures to accelerate domain-specific deductive synthesis systems. LNCS 1559 (1999) 61–82
16. Waldmann, U., Prevosto, V.: SPASS+T. In: ESCoR. (2006) 18–33
17. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: On a rewriting approach to satisfiability procedures: Extension, combination of theories and an experimental appraisal. In Gramlich, B., ed.: FroCos. Volume 3717 of Lecture Notes in Computer Science., Springer (2005) 65–80
18. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. Technical report, Microsoft Research (2007) to appear.
19. Aït-Kaci, H.: Warren's abstract machine: a tutorial reconstruction. MIT Press, Cambridge, MA, USA (1991)
20. Voronkov, A.: The anatomy of vampire implementing bottom-up procedures with code trees. J. Autom. Reasoning **15**(2) (1995) 237–265
21. Riazanov, A., A.Voronkov: Vampire 1.1 (system description). In: IJCAR '01. LNAI 2083 (2001) 376–380
22. Graf, P., Meyer, C.: Advanced indexing operations on substitution trees. In McRobbie, M.A., Slaney, J.K., eds.: CADE. LNCS 1104, Springer (1996) 553–567
23. Ganzinger, H., Nieuwenhuis, R., Nivela, P.: Context trees. In Goré, R., Leitsch, A., Nipkow, T., eds.: IJCAR'01. LNCS 2083, Springer (2001) 242–256
24. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7) (1970) 422–426
25. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970, Springer-Verlag (1983) 466–483
26. Leino, K.R.M., Musuvathi, M., Ou, X.: A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover. In: TACAS 05. LNCS 3440 (2005)
27. Ranise, S., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2006)