Efficient E-Matching for SMT Solvers

Leonardo de Moura and Nikolaj Bjørner

Quantifier reasoning

Motivation

Program verification problems often require dealing with quantified subformulas. [3]

Quantifier reasoning

$$P(f(42)) \wedge \forall x.P(f(x))) \Rightarrow x < 0$$

Quantifier reasoning

$$P(f(42)) \wedge \forall x.P(f(x))) \Rightarrow x < 0$$

Instantiate $P(f(x))$ *with the substitution* $[x := 42]$

$$P(f(42)) \wedge (\forall x.P(f(x))) \Rightarrow x < 0) \wedge (P(f(42)) \Rightarrow 42 < 0)$$

Quantifier reasoning

$$P(f(42)) \wedge \forall x.P(f(x))) \Rightarrow x < 0$$

Instantiate $P(f(x))$ *with the substitution* $[x := 42]$

$$P(f(42)) \wedge (\forall x.P(f(x))) \Rightarrow x < 0) \wedge (P(f(42)) \Rightarrow 42 < 0)$$

Apply Modus Ponens

$$42 < 0$$

Quantifier reasoning

Problem 1

How to figure out which instances are going to be useful?

Quantifier reasoning

Problem 1

How to figure out which instances are going to be useful?

Solution

Identify subterms occurring in the quantified formula (*patterns*), and only add instances that make those subterms equal to ground subterms that are currently being considered in the proof.

Quantifier reasoning

Problem 1

How to figure out which instances are going to be useful?

Solution

Identify subterms occurring in the quantified formula (*patterns*), and only add instances that make those subterms equal to ground subterms that are currently being considered in the proof.

Example

In the previous example, one such pattern is $P(f(x))$, which when instantiated with $[x := 42]$, yields its corresponding ground subterm $P(f(42))$.

Quantifier reasoning

Problem 2

Syntactic equality is oftentimes not enough. Consider eg. the following proposition:

$$a = f(42) \wedge P(a) \wedge \forall x.P(f(x))) \Rightarrow x < 0$$

There are no instances of $P(f(x))$ that matches the ground term $P(a)$, ie. there is no substitution for x that make $P(f(x))$ syntactically equal to $P(a)$.

Quantifier reasoning

Problem 2

Syntactic equality is oftentimes not enough. Consider eg. the following proposition:

$$a = f(42) \wedge P(a) \wedge \forall x.P(f(x))) \Rightarrow x < 0$$

There are no instances of $P(f(x))$ that matches the ground term $P(a)$, ie. there is no substitution for x that make $P(f(x))$ syntactically equal to $P(a)$.

Solution

Use the congruence closure of the equality relation induced by the current context.
For example, in the context $P(a), a = f(42)$, the substitution $[x := 42]$ makes $P(f(x))$ equal to $P(a)$, since $P(a) \simeq P(f(42))$.

Quantifier reasoning

Problem 3

How to identify the set of patterns for a given formula?

Quantifier reasoning

Problem 3

How to identify the set of patterns for a given formula?

Solution

- Heuristics
- Ask the user to provide the patterns

Quantifier reasoning

Problem 4

How to identify the substitutions that make the pattern equal to some ground term?

Quantifier reasoning

Problem 4

How to identify the substitutions that make the pattern equal to some ground term?

Solution

Apply an E-matching algorithm.

E-Matching algorithm (overview)

Input

- A binary relation E over terms
- A ground term t
- A pattern p

Output

The set of substitutions θ, modulo E, over the variables in p, such that

$$E \models t \simeq \theta(p)$$

E-matching algorithm

Definitions

- Σ is set of function symbols, called the *signature*
- V is the set of variables
- Each function symbol f has an arity, $arity(f)$
- An f-application is a term of the form $f(t_1, \ldots, t_n)$
- $T(\Sigma, V)$ is the set of terms
- $T(\Sigma, \varnothing)$ is the set of ground terms
- The set of non ground terms is called *patterns*
- A substitution β is a mapping from variables to ground terms
- A binary relation E over $T(\Sigma, \varnothing)$

E-graph

- An *E-graph* is a data-structure which maintains the *congruence closure* of a binary relation $E = \{(t_1, t'_1), (t_2, t'_2), \ldots\}$.
- This binary relation is given incrementally with the $union(t_1, t'_1)$ operation.
- Each equivalence class generated by the congruence relation has a *representative*.

E-graph

Operations

For each term t in the *E-graph*:

- $find(t)$ denotes the representative of the equivalence class that contains t
- $class(t)$ denotes the equivalence class that contains t
- $parents(f)$ denotes the set of terms $f(\ldots, t', \ldots)$ in the E-graph such that $t' \simeq t$
- $parents_f(t)$ is a subset of $parents(t)$ which contains only f-applications
- $parents_{f.i}(t)$ is a subset of $parents_f(t)$ which contains only f-applications where the i-th argument t_i is s.t. $t_i \simeq t$
- ...

Quantifier reasoning with E-matching

- Semantically, the formula

$$\forall x_1, \ldots, x_n . F$$

is equivalent to

$$\bigcup_{\beta} \beta(F)$$

where β ranges over all substitutions over the x's.

Quantifier reasoning with E-matching

- Semantically, the formula

$$\forall x_1, \ldots, x_n . F$$

  is equivalent to

$$\bigcup_{\beta} \beta(F)$$

  where β ranges over all substitutions over the x's.

- We now want to select from this set, the substitutions that are relevant to the conjecture.

Quantifier reasoning with E-matching

- Semantically, the formula

$$\forall x_1, \ldots, x_n . F$$

  is equivalent to

$$\bigcup_{\beta} \beta(F)$$

  where β ranges over all substitutions over the x's.

- We now want to select from this set, the substitutions that are relevant to the conjecture.

- To do so, we select non ground terms p from F as *patterns*, and consider $\beta(F)$ to be relevant whenever $\beta(p)$ is in the *E-graph.*

E-matching algorithm

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

E-matching algorithm

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

The abstract matching procedure returns all substitutions that E-match a pattern p with term t. That is, if $\beta \in match(p, t, \varnothing)$ then $E \models \beta(p) = t$.

Efficient E-matching

Complexity

E-matching is in theory NP-hard, and the number of matches can be exponential in the size of the E-graph.

In practice

The cost is dominated by:

- Matching a set of patterns against terms
- Compute sets of patterns that can generate new matches

Solution

Compile patterns into code, and introduce further optimizations.

E-matching abstract machine

Memory

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

E-matching abstract machine

Instructions

- init
- *bind*
- check
- compare
- choose
- *yield*
- *backtrack*
- choose-app

Semantics

init$(f, next)$	assuming $reg[0] = f(t_1, \ldots, t_n)$ $reg[1] := t_1; \ldots; reg[n] := t_n$ $pc := next$
bind$(i, f, o, next)$	$push(bstack,$ choose-app$(o, next, apps_f(reg[i]), 1))$ $pc :=$ backtrack
check$(i, t, next)$	**if** $find(reg[i]) = find(t)$ **then** $pc := next$ **else** $pc :=$ backtrack
compare$(i, j, next)$	**if** $find(reg[i]) = find(reg[j])$ **then** $pc := next$ **else** $pc :=$ backtrack
choose$(alt, next)$	**if** $alt \neq nil$ **then** $push(bstack,\ alt)$ $pc := next$
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**
choose-app$(o, next, s, j)$	**if** $\|s\| \geq j$ **then** **let** $f(t_1, \ldots, t_n)$ be the j^{th} term in s. $reg[o] := t_1; \ldots; reg[o + n - 1] := t_n$ $push(bstack,$ choose-app$(o, next, s, j + 1))$ $pc := next$ **else** $pc :=$ backtrack

Figure 1: Semantics of abstract machine instructions

Example

The pattern

$$f(x_1, g(x_1, a), h(x_2), b)$$

is compiled to into the following *labeled instructions*:

Label	Instruction
n_0	init(f, n_1)
n_1	check$(4, b, n_2)$
n_2	bind$(2, g, 5, n_3)$
n_3	compare$(1, 5, n_4)$,
n_4	check$(6, a, n_5)$
n_5	bind$(3, h, 7, n_6)$
n_6	yield$(1, 7)$

E-matching abstract machine

Usage

To compute $match(p, t, \varnothing)$, where p is eg. the pattern on the previous slide:

- Store t in $reg[0]$
- Set $pc = n_0$
- Execute the instruction stored in pc
- Collect the yielded substitutions

Efficient E-matching

To further improve efficiency, the paper introduces two novel techniques:

- E-matching code trees
- Inverted path index

E-matching code trees

Problem

Oftentimes, patterns have some shared structure, and the solver wastes time repeatedly matching against these subpatterns.

Example

- $f(x, g(a, y))$
- $f(x, g(x, y))$
- $f(h(x, y), b)$
- $f(h(x, g(x, y)), b)$

E-matching code trees

Solution

- Minimize the time spent on matching similar patterns by combining their code sequences in a *code tree*
- The matching work common to multiple patterns is *factored out*
- Whenever the DPLL(T) engine asserts a quantified formula, add its patterns to the code tree. Remove them when the engine backtracks.

E-matching code trees

$$P = \{f(x, g(a, y)), f(x, g(x, y)), f(h(x, y), b), f(h(x, g(x, y)), b)\}$$

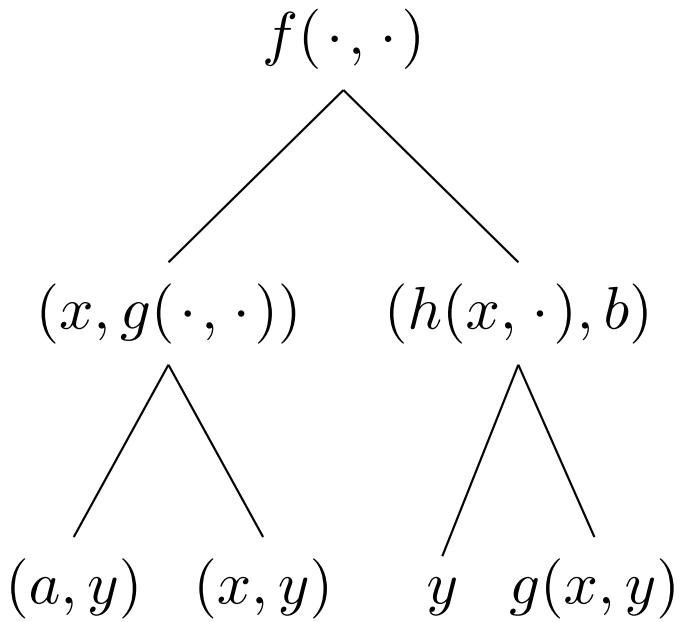


Figure 2: Code tree for P

E-matching code trees

$$P = \{f(x, g(a, y)), f(x, g(x, y)), f(h(x, y), b), f(h(x, g(x, y)), b)\}$$

$\mathsf{init}(f, n_1)$

$n_1 : \mathsf{choose}(n_9, n_2),\ n_2 : \mathsf{bind}(2, g, 3, n_3)$

$n_3 : \mathsf{choose}(n_6, n_4),\ n_4 : \mathsf{check}(3, a, n_5),\ n_5 : \mathsf{yield}(1, 4)$

$n_6 : \mathsf{choose}(nil, n_7),\ n_7 : \mathsf{compare}(1, 3, n_8),\ n_8 : \mathsf{yield}(1, 4)$

$n_9 : \mathsf{choose}(nil, n_{10}),\ n_{10} : \mathsf{check}(2, b, n_{11}),\ n_{11} : \mathsf{bind}(1, h, 5, n_{12})$

$n_{12} : \mathsf{choose}(n_{14}, n_{13}),\ n_{13} : \mathsf{yield}(5, 6)$

$n_{14} : \mathsf{choose}(nil, n_{15}),\ n_{15} : \mathsf{bind}(6, g, 7, n_{16}),\ n_{16} : \mathsf{compare}(5, 7, n_{17}),\ n_{17} : \mathsf{yield}(5, 8)$

Figure 3: Code tree for P

Incrementality

- Say we have a term $f(a, b)$, and a pattern $f(g(x), y)$
- At this stage, there is no way to instantiate $f(g(x), y)$ so that it matches $f(a, b)$
- At some point, the operation $union(a, g(c))$ is executed
- The E-graph now tells us that $f(a, b) \simeq f(g(c), b)$
- It thus becomes possible to instantiate $f(g(x), y)$ with $[x := c, y := b]$ to match $f(g(c), b)$.

Incrementality

Problem

How to identify new terms and patterns that become relevant for matching?

Incrementality

Problem

How to identify new terms and patterns that become relevant for matching?

Solution

Introduce an *inverted path index*.

Inverted path index

pc-pair

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:

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

Inverted path index

Inverted path string

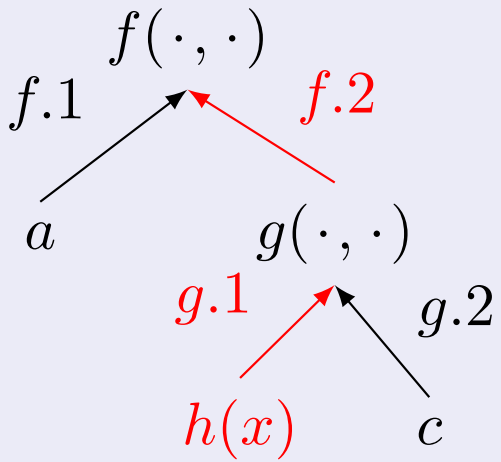
An *inverted path string* over a signature Σ is either:

- the empty string ϵ
- $f.i.\pi$, where π is an inverted path string, $f \in \Sigma$, and $i \in i\mathbb{N}$.

Inverted path index

Inverted path string

The inverted path string $g.1.f.2$ is a path to term $f(a, g(h(x), c))$ from subterm $h(x)$.



Inverted path index

Construction

Given a set of patterns P which share the same set of pc-pairs, one of those pc-pair (f, g), and a pattern $p \in P$:

- $\Pi(p, (f, g))$ denotes the set of inverted path strings from every relevant g-application in p
- The *inverted path index* $\tau(P, (f, g))$ is a trie over $\bigcup_{p \in P} \Pi(p, (f, g))$.

Inverted path index

$$P = \{f(f(g(x), a), x), h(c, f(g(y), x)), f(f(g(x), b), y), f(f(a, g(x)), g(y))\}$$

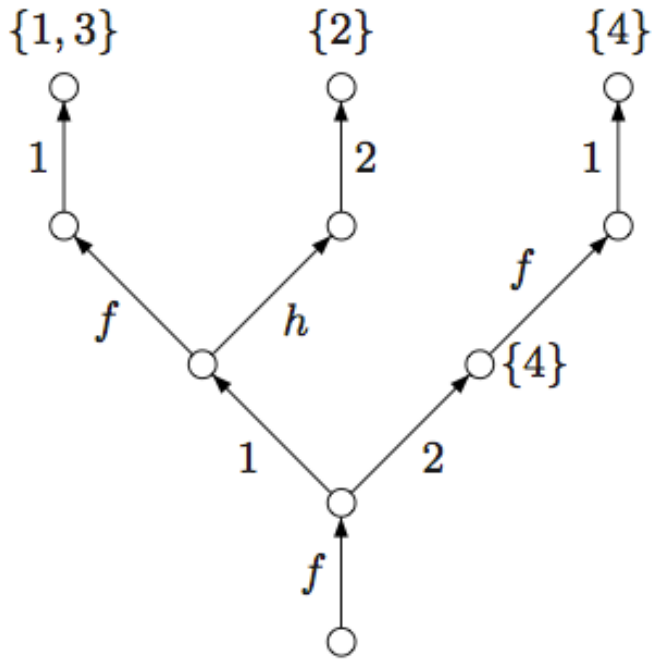


Figure 4: Inverted path index for *pc-pair* (f, g) and patterns P

Inverted path index

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

Figure 5: Algorithm to compute U

Reminder

$parents_{f.i}(t)$ is the set of of f-applications where the i-th argument t_i is congruent to t

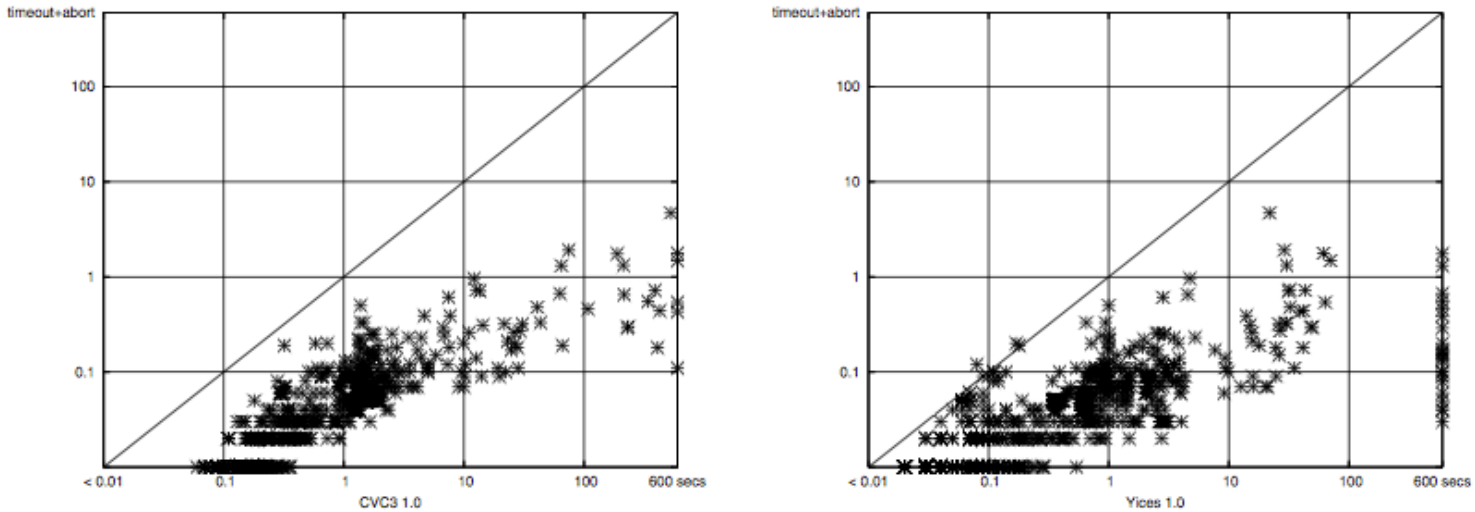
Inverted path index

Usage (?)

Whenever the E-graph is updated, ie. after executing $union(t_1, t_2)$:
For each *pc-pair* (f, g) relevant to t_1 and t_2,

- Define P to be the set of patterns with *pc-pair* (f, g)
- Compute the set $U = collect(\tau(P, (f, g)), \{t_1, t_2\})$ of newly relevant patterns-terms pairs using the inverted path index
- For each $(Q, T) \in U, p \in Q, t \in T$, execute $match(p, t, \varnothing)$, and collect the yielded substitutions

Benchmarks

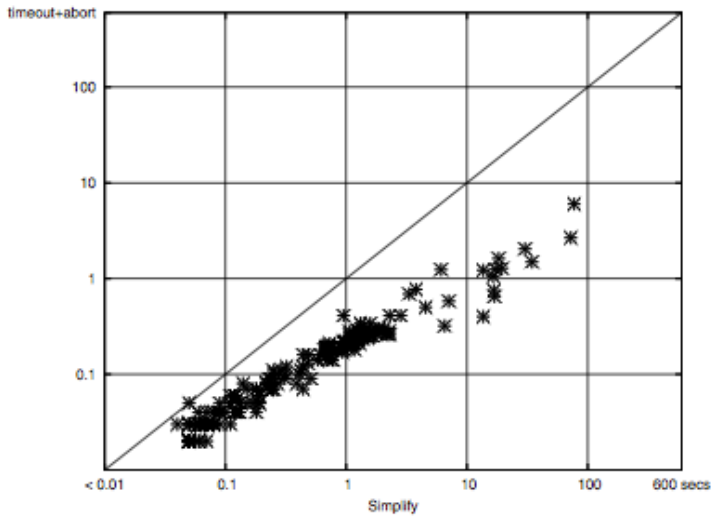


(a) Z3 vs. CVC3 1.0

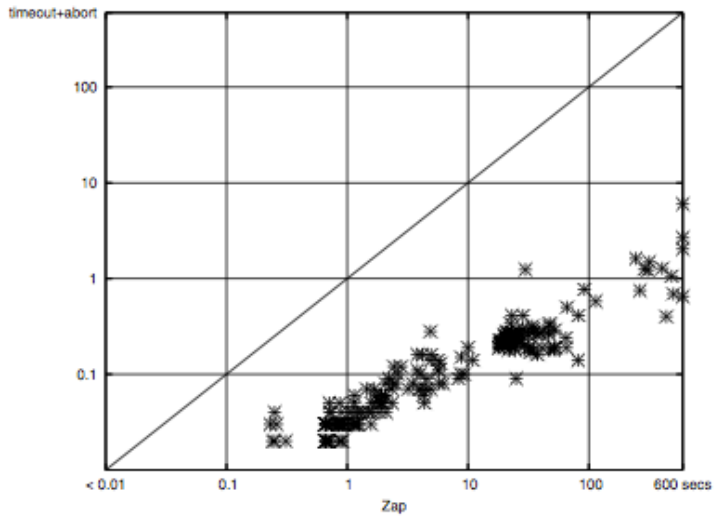
(b) Z3 vs. Yices 1.0

Figure 6: SMT-LIB benchmarks

Benchmarks



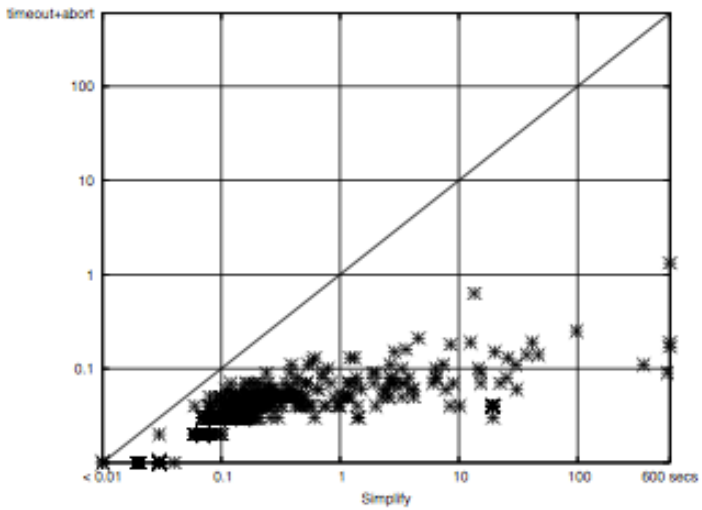
(a) Z3 vs. Simplify



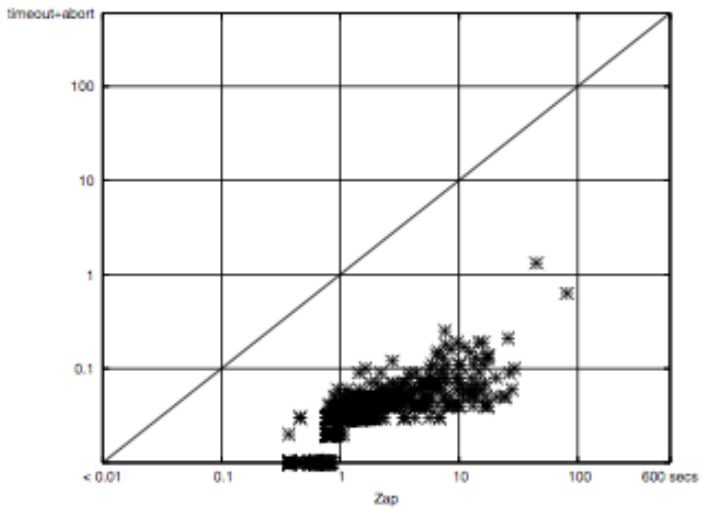
(b) Z3 vs. Zap 2.0

Figure 7: ESC/Java benchmarks

Benchmarks



(a) Z3 vs. Simplify



(b) Z3 vs. Zap 2.0

Figure 8: Boogie benchmarks

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**

Figure 9: Experimental results: summary

Contributions

- An *abstract machine* for E-matching
- *E-matching code trees*, which can efficiently handle matching a term against a large set of patterns simultaneously
- *Inverted path indexing*, which narrowly and efficiently finds a superset of terms that will match a set of patterns

Further work

Use *context trees* for additional sharing

A context tree is a data-structure where not only prefix terms but also common subterms can be shared, even if they occur below different function symbols. [2]

References

[1] Detlefs, D. et al. 2005. Simplify: A theorem prover for program checking. *J. ACM*. 52, 3 (May 2005), 365–473.
[2] Ganzinger, H. et al. 2001. Context trees. *Automated reasoning: First international joint conference, ijcar 2001 siena, italy, june 18–22, 2001 proceedings*. R. Goré et al., eds. Springer Berlin Heidelberg. 242–256.
[3] Moskal, M. et al. 2008. E-matching for fun and profit. *Electron. Notes Theor. Comput. Sci.* 198, 2 (May 2008), 19–35.
[4] Moura, L. de and Bjørner, N. 2007. Efficient E-matching for SMT solvers. *Automated deduction – cade-21: 21st international conference on automated deduction bremen, germany, july 17-20, 2007 proceedings*. F. Pfenning, ed. Springer Berlin Heidelberg. 183–198.