EasyChair Preprint

№ 10510

Lazy and Eager Patterns in High-Performance Automated Theorem Proving

Stephan Schulz

July 8, 2023

Lazy and Eager Patterns in High-Performance Automated Theorem Proving

Stephan Schulz

DHBW Stuttgart
Stuttgart, Germany
schulz@eprover.org

Abstract

Eager maintenance of invariants is often the first approach to implement high-performance data structures. We present a number of examples from the evolving implementation of the theorem prover E to demonstrate that a more relaxed, lazy implementation can have advantages for simplicity, performance, or both.

1 Introduction

Most high-performance automated theorem provers are large, complex programs, utilizing efficient data structures and tight programming. In our experiments, many implementations use eager techniques, i.e. the programmer tries to immediately resolve all implications of an action and restore the strongest possible invariants on the affected data structures. Examples from the history of E [5, 8] are eager shared rewriting, eager term cell garbage collection, and eager orphan removal. Over time, we have found that a more relaxed, lazy approach can result in simpler, easier to maintain, less brittle, and often even more efficient code. In this paper, we discuss the basic principle, the examples mentioned above, and possible future applications of the same lazy paradigm within E.

E is a saturating theorem prover for first-order logic with equality[1]. It is based on the superposition calculus with rewriting and subsumption. The most challenging part is the implementation of the actual given-clause loop, which saturates a set of clauses. It is based on a separation of clause into processed clauses (between which both generating and simplifying inferences have been performed), and unprocessed clauses, which may be partially simplified, but have not yet been used for generating inferences. The algorithm heuristically picks an unprocessed clause, normalises it, back-simplifies the processed clauses with this *given clause*, and then performs generating inferences between the given clause and the processed clauses. The resulting clauses are added to the set of unprocessed clauses, the given clause to the set of processed clauses. The process terminates when the system runs out of unprocessed clauses, or generates the empty clause as an explicit witness of a proof by contradiction.

Some of the critical aspects of the implementation are listed below:

- The set of clauses and in particular of terms grows very fast. Historically, running out of memory was a major problem. Modern provers often use *shared terms* to reduce the memory burden.
- The most important simplification techniques are (unconditional) rewriting and subsumption. In both cases, we need to efficiently find clauses that can simplify a particular clause, or can be simplified by it. Most high-performance provers use one or more indexing techniques for this [4, 10].

[1] The latest versions also support higher-order logic, but this is not a significant difference for our discussion

- The set of unprocessed clauses will grow very large quickly. The prover still needs to pick heuristically promising clauses from it, usually by more than one criterion.

In the following we will look at some of these aspects and how lazy programming can lead to simpler, clearer and even more efficient implementations.

2 Tales from the Great War

First, we discuss situations in which the change from an eager to a lazy solution has already helped us to produce simpler, cleaner code.

2.1 Shared Rewriting

E was originally developed with the idea of not only shared terms, but also shared rewriting. A rewrite performed once should affect all occurrences of that particular subterm in all clauses. This was achieved by term cells carrying pointers not only to subterms, but also to super-terms (and to occurrences in clauses). Changes were propagated through the shared term structure. Implementation of this scheme was, to say the least, painful in the extreme. But of course, we expected a big payoff. However, when we compare E's shared rewriting to Waldmeister's [1] plain rewriting, we found not only that Waldmeister was about 5 times faster (which we expected, due to tighter coding of the more specialised system), but also that this advantage was more or less constant and independent of the effective sharing factor in E [2]. Analysis showed that management of the superterm pointers incurs about the same amount of extra effort (in the Big-Θ sense) than repeated rewriting.

We then (for release 0.71, released in late 2002) moved to shared terms with cached rewriting. Instead of eagerly pushing changes to super-terms and clauses, we just add a rewrite link to the affected subterm. Only when a term or clause is due to be simplified anyways do we check if we can use such a previously established link instead of searching for suitable rewrite rules *ab initio*. While we did not collect any comparative performance data back then, we can say that this system is so much simpler to implement, maintain, and modify, that it is unquestionably a much superior choice. It also enabled a number of subsequent simplifications of the code base. As an example, we don't need to special-case maximal terms (i.e. terms that are maximal in maximal literals of any clause) in the term bank, as they are now only accessed via the clause objects, where their role is implicitly clear.

2.2 Garbage Collection

At the same time we moved from eager to cached shared rewriting, we also changed to a new kind of garbage collection for our shared terms. Shared terms are organised in *term bank*. The original implementation used reference counting garbage collection, and would eagerly delete terms with a reference count of zero (and then propagate this change to subterms). While sound in principle, this required the system to explicitly free the reference whenever a clause was changed or deleted - a task that quickly became tedious. It also required us to maintain a reference counter in each term cell.

Our new implementation uses a mark-and-sweep garbage collector. Not only is this much simpler, it also frees developers from explicitly freeing references. Moreover, if terms are recreated during the proof search, there is a good chance that the old copy, including the cached rewrite links, are still in the term bank and can be reused. All this makes handling terms much easier, and has allowed us to maintain a zero-leak policy with respect to memory management.

2.3 Orphan Removal

Orphans are unprocessed clauses that have been generated from parent clauses of which at least one was later shown to be redundant. That automatically makes the orphan redundant, too. Removing such redundant clauses significantly reduces the search space and significantly speeds up the search process. In the first implementation of E, every clause maintained a list of its children, and every child maintained pointers to its parent clauses. When a (processed) clause was back-simplified or otherwise shown to be redundant, it actively culled its dependend children (which, in turn, would inform potential other parents of their early demise). The system worked reasonably well, but maintaining all relations between parents and children became quite cumbersome.

When we started to record internal proof objects with E 1.8 [7, 9], all clauses which had produced offspring had to be archived forever, anyways. Since the parent clauses never vanished, we could move to a lazy orphan removal scheme: Only when an unprocessed clause was selected for processing did we check the status of the parents. When one of them was marked as back-simplified, we could discard the child and select the next unprocessed clause. For this, we only need to link from the child to each parent - a link that is needed for proof reconstruction anyways. But we could get rid of the links from parents to children, and the complex notification system. Again, this resulted in a significant simplification in the complexity of the code.

3 Science Fiction

In this section we suggest some as yet unused opportunities we see to improve the system using lazy patterns.

3.1 Index Optimisation

The most central and first implemented index used by E is a perfect discrimination tree index [3] used for forward rewriting. The tree indexes (potential) rewrite rules (i.e. equational unit clauses) via their potential left hand sides (maximal terms). The query is a single term, the index returns those of these left hand sides that match this query, and, of course, the associated clauses. If a found unit clause is or can be oriented, it can be used to simplify the query term. A discrimination tree basically represents a set of terms as a trie of its flat string representation, i.e. it is a branching tree, where each subsequent function- or variable symbol selects one subbranch. A full term is represented by the path from the root to a leaf node.

This is one of the standard index data structures for this purpose. However, E uses two refinements: Size and age constraints. The first is based on the fact that instantiation can only increase the (symbol counting) size of a term. In other words, a term can never be matched by a bigger term. Thus, if we know the maximal size of any term at the leaves of a subtree, and this size is greater than that of the query term, none of the indexed terms can match (indeed, since we use perfect discrimination trees and perform variable bindings during the query execution, we can even compute somewhat stronger dynamic constraints). In order to use this feature, we must store the maximal size of all indexed subterms at the root of each subtree. Currently, we use an eager algorithm that always maintains this value. When a new term is inserted into a subtree, we check if that term's size is bigger than the currently stored value. If so, we update that value. When a term is deleted from the index, we check if its size is the maximal size of terms in that subtree, and if so, we aggressively recompute the new value from all the remaining terms (or branches in inner nodes). While this is easy, it also comes

at a certain cost. We believe that a lazy algorithm can make things simpler. We simply add an *invalid* marker (e.g. -1) to the set of possible size values. When only dealing with valid values, insertion works as before. When encountering an invalid node, we simply don't update the value. When deleting a maximal weight term from a subtree with a valid weight, we simply replace the local weight by the invalid marker. Only when we encounter an invalid value during a query do we recompute and set the correct value. The advantage is, of course, that we do not recompute values not needed, in particular not if multiple deletions follow each other (or are only interleaved with queries that do not touch *invalid* weight branches).

While we have explained the approach with perfect discrimination trees and size constraints, it also translates to fingerprint indices [6] for back-simplification, and to other constraint types (e.g. age constraints - a term in normal form with respect to the rewrite system at time t cannot be rewritten by any older clause).

3.2 Clause Selection

Our last example is the selection of the next given clause - the central choice point of the saturation algorithm. For this, clauses are evaluated, and organised in a priority queue. Clauses are processed in order of increasing heuristic weight. This should be a perfect application for a min-heap. Insertion, the most frequent operation, is $O(1)$. Moreover, new clauses skew heavy, so even the constants are small. Removing the smallest clause, the less frequent operation, is still $O(\log n)$. However, most provers support more than one of these queues, and E indeed supports an arbitrary number of those queues, from which clauses are picked in a weighted round-robin scheme. Removing an entry from an arbitrary position in a priority queue is not easy nor elegant, and a naive implementation is $O(n)$. Therefore, the current implementation, which eagerly removes picked clauses from all queues, uses splay trees [11]. Splay trees are amortised $O(\log n)$ for insertion and deletion. The also have more memory overhead, since each evaluation needs two children pointer, while heaps can be embedded in arrays without this overhead.

Here, we can once again use the lazy pattern - when we remove a the best clause from any queue, we simply mark the entry as obsolete. If we get an obsolete entry from a queue, we discard it (thus lazily removing it from this queue, too), and request the next one. This enables us to use the optimal performance of heaps with a simple and elegant algorithm.

4 Conclusion

We have show a number of actual and potential programming situations in the implementation of automated theorem provers were moving from an eager to a lazy implementation can significantly reduce code complexity. We hope these ideas inspire other developers too keep the lazy implementation paradigm in mind.

References

[1] B. Löchner and Th. Hillenbrand. A Phytography of Waldmeister. *Journal of AI Communications*, 15(2/3):127–133, 2002.

[2] B. Löchner and S. Schulz. An Evaluation of Shared Rewriting. In H. de Nivelle and S. Schulz, editors, *Proc. of the 2nd International Workshop on the Implementation of Logics*, MPI Preprint, pages 33–48, Saarbrücken, 2001. Max-Planck-Institut für Informatik.

[3] W.W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[4] R. Nieuwenhuis, Th. Hillenbrand, A. Riazanov, and A. Voronkov. On the Evaluation of Indexing Techniques for Theorem Proving. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 257–271. Springer, 2001.

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

[6] 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.

[7] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

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

[9] Stephan Schulz and Geoff Sutcliffe. Proof generation for saturating first-order theorem provers. In David Delahaye and Bruno Woltzenlogel Paleo, editors, *All about Proofs, Proofs for All*, volume 55 of *Mathematical Logic and Foundations*, pages 45–61. College Publications, London, UK, January 2015.

[10] R. Sekar, I.V. Ramakrishnan, and A. Voronkov. Term Indexing. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 26, pages 1853–1961. Elsevier Science and MIT Press, 2001.

[11] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.