Checking Data Structure Properties Orders of Magnitude Faster

Emmanouil Koukoutos and Viktor Kuncak

EPFL, Lausanne, Switzerland
{emmanouil.koukoutos,viktor.kuncak}@epfl.ch

Abstract. Executable formal contracts help verify a program at runtime when static verification fails. However, these contracts may be prohibitively slow to execute, especially when they describe the transformations of data structures. In fact, often an efficient data structure operation with $O(\log(n))$ running time executes in $O(n \log(n))$ when naturally written specifications are executed at run time.

We present a set of techniques that improve the efficiency of run-time checks by orders of magnitude, often recovering the original asymptotic behavior of operations. Our implementation first removes any statically verified parts of checks. Then, it applies a program transformation that changes recursively computed properties into data structure fields, ensuring that properties are evaluated no more than once on a given data structure node. We present evaluation of our techniques on the Leon system for verification of purely functional programs.

1 Introduction

Static verifiers can demonstrate program correctness for any given input. However, their limitations prevent them from proving complex programs. Runtime verification can be of great help in such circumstances. Unfortunately, contracts that are good for static verification are often expensive to check at runtime, and may even degrade program performance asymptotically.

Related Work. There have been some attempts to mitigate the performance penalty of runtime checks. Shankar and Bodík [7] present DITTO, an automatic incrementalizer for imperative data structure invariant checking. The system memoizes results of runtime checks for data structures, and recomputes them only when the data structure is mutated, rather than every time it is accessed. Memoization itself is first proposed by Michie [5]. Hughes [4] introduces lazy memo-functions, which optimize memoization by computing the results of the memoized functions lazily. Memoization (or tabling) has also been included as a built-in feature in XSB and other variants of Prolog [9], providing both theoretical and practical benefits to performance and termination of Prolog programs.

Another popular strategy for optimizing runtime checks is partially evaluating checks ahead of time, i.e. running a static verification step before executing the program, in order to simplify or completely remove runtime checks. This idea

```
sealed abstract class Tree
case class Leaf() extends Tree
case class Node(left : Tree, value : Int, right: Tree) extends Tree
def insert(t: Tree, e : Int) : Tree = {
  require(isBST(t))
  t match {
    case Leaf() ⇒ Node(Leaf(), e, Leaf())
    case Node(l,v,r) ⇒ if (e == v) t
      else if (e < v) Node(insert(l,e), v, r)
      else Node(l, v, insert(r,e))}
} ensuring (res ⇒ isBST(res))

def isBST(t:Tree) : Boolean = t match {
  case Leaf() ⇒ true
  case Node(l,v,r) ⇒ isBST(l) && isBST(r) && treeMax(l) < v && v < treeMin(r) }
```

Fig. 1. Binary Search Tree

```
case class TreeFields(isBST: Boolean, treeMin : Int, treeMax : Int)
sealed abstract class Tree
case class Leaf(treeFields : TreeFields) extends Tree
case class Node(left : Tree, value : Int, right: Tree, treeFields : TreeFields) extends Tree
def makeNode(left : Tree, value : Int, right: Tree) = Node(left, value, right, {
  val lmin = left.treeFields.min; val lmax = left.treeFields.max
  val rmin = right.treeFields.min; val rmax = right.treeFields.max
  val thisBST = left.treeFields.isBST && right.treeFields.isBST && lmax < v && v < rmin
  TreeFields(thisBST, min3(lmin,value,rmin), max3(lmax,value,rmax)) }
def isBST(t:Tree) : Boolean = t.treeFields.isBST
```

Fig. 2. Binary Search Tree with Memoization

has been applied to partially evaluate finite-state properties [2] as well as for dynamically typed languages and languages with expressive type systems [3].

Contributions. In this paper, we show that a particular, predictable form of memoization, which introduces extra fields into data structures, can substantially improve the performance of runtime checks that remain after static verification attempts. Our system works for a purely functional subset of Scala recognized by the Leon verification tool [1], [8]. It provides executable formal contracts in the form of pre- and postconditions of functions, and supports algebraic data types (ADTs). Our implementation and the benchmarks we used to evaluate it are available at `https://github.com/manoskouk/leon/tree/memoization`.

Example. Consider the example in Fig. 1, which defines a tree datatype along with a decorated insert operation in PureScala. **require** denotes a precondition, whereas **ensuring** denotes a postcondition that takes an anonymous function that applies to function result. isBST is a function used as a specification, denoting that its argument is indeed a binary search tree. treeMin and treeMax are full tree traversals, so isBST would also need to traverse the whole tree when executed. If executed directly as written, each recursive call of insert calls isBST, which makes insert on unbalanced trees quadratic instead of linear. To avoid such costly computation without losing any information, we add extra fields into Tree to denote the result of isBST, treeMin and treeMax for each node. Fig. 2 sketches such transformed version of the data structure. The information that was previously computed using recursive functions is now available with a field lookup. We use constructor functions such as makeNode to compute the additional fields when creating a node, using a constant amount of additional work. The next sections present and evaluate an automated transformation that performs such rewriting.

2 Our Approach

In our system, memoization and static verification jointly reduce the cost of runtime checks.

2.1 Memoizing Fields for Formal Contracts

Intuitively, we memoize whatever the program's formal contracts need, and use the data structure itself as the storage space. A function is eligible to be memoized, if 1) it is called (directly or indirectly) from a formal contract (otherwise its value is not needed for runtime checks), 2) it has a single argument of a class type, i.e. an ADT (to ensure that a single memoized field can indeed uniquely describe the result of the function for the object it is applied to), 3) it is recursive, possibly through mutual recursion (to make memoization worthwhile), 4) its return type is not the same as its argument type (as a heuristic to exclude storing large fields). Our system memoizes each function that fits the above criteria by turning it into an extra field. Each invocation of the function in the program is substituted with a field retrieval. Every instantiation of a class that was enhanced with extra fields is modified to initialize these fields correctly.

Memoizing further fields. The above memoization technique is not specific to runtime checks, but can also memoize bookkeeping fields for data structures that require them, such as AVL sub-tree heights. This saves the programmer the effort to prove that the memoized field matches a definition. Our system therefore supports an explicit annotation in the source code to memoize additional functions. We have used this functionality to simplify some of our benchmarks.

2.2 Utilizing Static Verification

Memoization, although useful on its own, does not yield the optimal results in isolation. This is because we often end up memoizing fields to monitor properties that have already been proven statically. This may have large associated cost, especially when complex properties are involved (e.g. the contents of a data structure).

Therefore, in our system, memoization is preceded by static verification. Each statically verified postcondition is removed from the program, and each precondition verified at a call site is removed from this call site. Additionally, all formal contracts expressed as conjuncts of simpler contracts are split and the conjuncts are verified separately, to remove as many checks as possible.

3 Evaluation

To demonstrate how this transformation can improve programs, we evaluated it on benchmarks available at `https://github.com/manoskouk/leon/tree/memoization`. We consider benchmarks where we perform a series of element insertions to a data structure, and benchmarks where we sort a list.

In Table 1, we compare the asymptotic running time bounds for the original version of each benchmark with the fully optimized version, where we have removed statically verified formal contracts and then applied memoization. This analysis demonstrates that our approach indeed restores the asymptotic bounds of programs in many cases.

Table 1. Asymptotic time bounds. [1]Per element insertion. [2]Amortized.

Benchmark	SortedList[1]	AVLTree[1]	RBTree[1]	AmQueue[1,2]	HeapSort	InsertionSort
Original	$O(n^2)$	$O(n \log n)$	$O(n)$	$O(n)$	$O(n^2 \log n)$	$O(n^3)$
Optimized	$O(n)$	$O(\log n)$	$O(n)$	$O(1)$	$O(n \log n)$	$O(n^2)$

To further confirm the results, we compiled and ran these benchmarks. The tests were compiled to JVM bytecode with the internal compiler of Leon. The input used was a sequence of pseudorandom numbers produced by a simple arithmetic function (using linear and *mod* operators). For each benchmark, we present four sets of measurements, corresponding to the original program, and the version of the program after applying each of our two techniques, isolated or together ("Original", "Memoized", "Static" and "Static+Memoized").

Note that the AVLTree and HeapSort benchmarks had the subtree height automatically memoized rather than manually in the original version, which makes the "Original" and "Static" versions slower; however, this influences asymptotically only the "Static" version of HeapSort (by a logarithmic factor), since in all other cases the performance penalty is at least matched by unverified checks. Also, we use an implementation of AmortizedQueue that uses the sizes of the front and back stacks to decide when it has to reverse the former onto the latter [6]. So our system memoizes the size of both stacks.

The results are presented in Fig. 3. Missing measurement points mean that the corresponding benchmark timed out with a timeout of 100 seconds.

Our techniques improved the performance of programs by orders of magnitude. In several of these benchmarks, all checks were removed statically; this is because we originally started from benchmarks that were used to show the

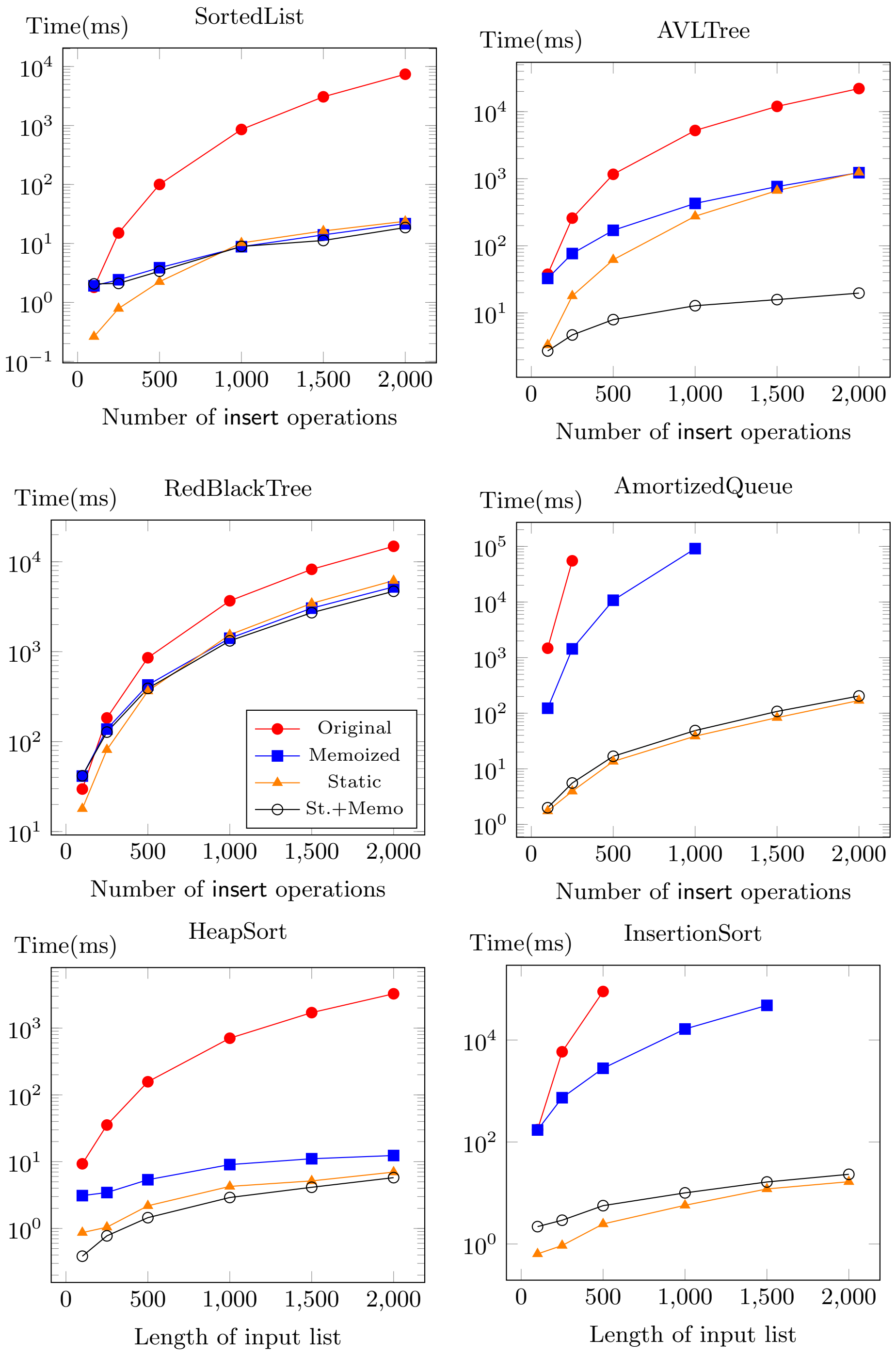

Fig. 3. Performance of programs before and after memoization

strengths of static verification of Leon [8]. In these cases, the memoized version without checks performs almost identically to the non-memoized one. It is also notable that the memoized version with all checks was always much better than the original one, even when tracking the contents of the data structures. This indicates that memoization can be useful in more difficult problems as well, where static verification fails to remove any contracts.

Comment on space usage. Although memoization improves the running time of programs, it has negative impact on space usage. For most of our benchmarks the increase is by a constant factor. This is in our opinion acceptable for JVM, where objects already have a large footprint. The only exception is the RBTree benchmark, where a set representing the content of each subtree had to be memoized, resulting in asymptotically increased space usage (about 100 times for input size 2000). In future versions of our system we will rule out memoization of such complex properties, or better exploit the opportunities for fine-grained sharing within memoized values, using techniques such as hash consing.

Conclusion. Overall, we have found that memoization provides orders of magnitude improvements in running time of benchmarks compared to directly executing formal contracts. It works well both in isolation, or in synergy with less predictable techniques of static verification.

References

1. Blanc, R.W., Kneuss, E., Kuncak, V., Suter, P.: An overview of the Leon verification system: Verification by translation to recursive functions. In: Scala Workshop (2013)
2. Bodden, E., Lam, P., Hendren, L.: Partially evaluating finite-state runtime monitors ahead of time. ACM Trans. Program. Lang. Syst. 34(2), 7:1–7:52 (2012)
3. Flanagan, C.: Hybrid type checking. In: Morrisett, J.G., Jones, S.L.P. (eds.) POPL, pp. 245–256. ACM (2006)
4. Hughes, J.: Lazy memo-functions. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 129–146. Springer, Heidelberg (1985)
5. Michie, D.: Memo functions and machine learning. Nature 218(5136), 19–22 (1968)
6. Okasaki, C.: Functional data structures. In: Launchbury, J., Sheard, T., Meijer, E. (eds.) AFP 1996. LNCS, vol. 1129, pp. 131–158. Springer, Heidelberg (1996)
7. Shankar, A., Bodik, R.: Ditto: automatic incrementalization of data structure invariant checks (in Java). ACM SIGPLAN Notices 42, 310–319 (2007)
8. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 298–315. Springer, Heidelberg (2011)
9. Swift, T., Warren, D.S.: Xsb: Extending Prolog with tabled logic programming. Theory and Practice of Logic Programming 12(1-2), 157–187 (2012)