How to Efficiently Process 2^{100} List Variations

Lukas Lazarek
University of Massachusetts Lowell
Lowell, Massachusetts, USA

Abstract

Variational execution offers an avenue of efficiently analyzing configurable systems, but data structures like lists require special consideration. We implement automatic substitution of a more efficient list representation in a variational execution framework and evaluate its performance in microbenchmarks. The results suggest that the substitution may offer substantial performance improvements to programs involving highly variational lists.

CCS Concepts • **Software and its engineering → Software configuration management and version control systems**; *Software reliability*;

Keywords Configurable Systems, Variational Execution, Testing

1 Configurable Systems

Configurable systems pose distinct challenges for program analysis and testing. The number of system configurations grow as much as exponentially with the number of configurable features, and each combination may produce unexpected behavior [6]. Such systems must therefore be analyzed with special consideration for configuration complexity, but this is non-trivial due to the number of configurations. Furthermore, serious bugs can be caused by configurations, such as the HeartBleed vulnerability caused by an often unnecessary but default configuration option [4]. Analyzing these systems therefore presents a significant verification and security problem.

2 Variational Execution

Variational execution is a program analysis technique that exploits sharing among configurations to enable reasoning across all configurations [5, 6, 11]. It is similar to symbolic execution, but it executes the program with concrete values distinguished by symbolic contexts. A *context* is a set of configurations under which executions are performed and data is stored, similar to a path condition [3]. Contexts may be represented by propositional formulas; for example, context $A \wedge B$ represents configurations where both features A and B are enabled. Executions and data are shared by collecting operations and values common to many configurations under a single context. Because most executions and data are redundant across configurations [5, 6, 11], this enables efficient analysis of highly configurable systems.

3 Variational Lists Cause Explosion

Although this strategy is general, naive representation of data structures can have extreme performance repercussions. Lists are a simple representative of these problems. A variational list can be naively implemented by storing a separate list for every (feature-dependent) variation of its elements. However, this representation grows exponentially when many of the list's entries are added under different contexts, such as when building a list of optional plugins that are active. As an illustration, consider the construction of the list below, where the first two elements are added under different contexts (as in CheckStyle or WordPress [6]) and the last two under context *True* (every configuration).
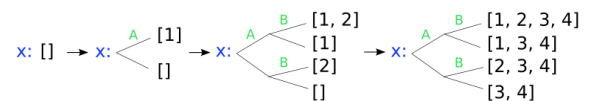

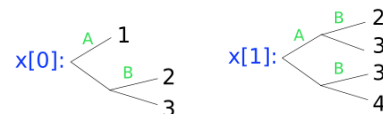**Figure 1.** Building a naive variational list.

Figure 2. Accessing elements of a naive variational list.

Iterating over this list (figure 1) is extremely expensive. When iterating, elements are obtained in sequence for all

configurations. The first element of a naive list is a choice between the first element of every list variation (see figure 2). Such choices make iteration over naive variational lists expensive. To analyze CheckStyle, for example, Meinicke et al. needed to rewrite its code to avoid adding list entries under many contexts [6].

4 Automatic CtxList Substitution

We contribute a fully automatic bytecode transformation that substitutes CtxList, a more efficient list implementation, for naive lists. CtxList represents variational lists as a plain list of optional entries with associated contexts, just like Walkingshaw et al.'s OList [11]. CtxList differs only in that it stores pairs of value and context, implicitly encoding the optional presence of elements by assuming that they only exist in their associated contexts. Figure 3 illustrates CtxList using the above example.

$$\text{x: } [] \rightarrow \text{x: } [1_A] \rightarrow \text{x: } [1_A, 2_B] \rightarrow \text{x: } [1_A, 2_B, 3_T, 4_T]$$

Figure 3. Building a CtxList.

Context differences between elements do not affect CtxList's size, unlike the naive list. To iterate over CtxList we obtain the elements as they appear in the list and execute the body in the context of the element being processed. Thus, instead of attempting to index elements according to the context of the loop, elements are accessed regardless of context and the loop body is conditioned on each element's context.

This behavior ultimately preserves the original list semantics, so it can be automatically substituted for naive lists. However, iteration loops must then be transformed to perform CtxList's different method of iteration. Similar to loop unrolling [1], iteration loops must be rewritten transparently to the programmer. We modified a Java variational execution framework to automatically perform this transformation. The framework rewrites the bytecode of classes at load-time to execute the program variationally. During rewriting, instantiations of LinkedList and ArrayList are replaced with CtxList, and loops are identified using control flow analysis [1] and rewritten. [1]

5 Preliminary Evaluation

The CtxList replacement sought to improve the memory consumption and performance of processing lists variationally. To that end, we performed micro-benchmarks of the memory consumption and performance of CtxList for eleven common list operations with elements under a varying number of different contexts. One of those benchmarks performs a simplified representation of the CheckStyle behavior problematic for the naive list (storing elements under many different contexts).
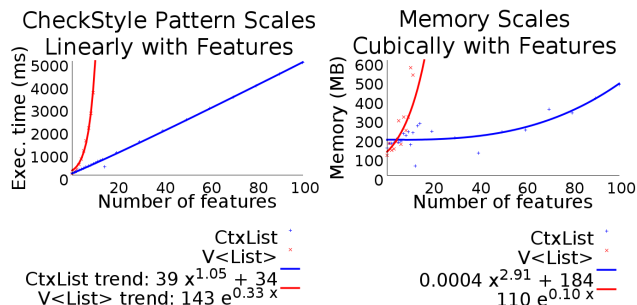

Figure 4. Benchmark results showing memory consumption and execution time for CheckStyle behavior.

Figure 4 shows that where the naive list's memory and execution time grow exponentially with features, CtxList's performance scales linearly for the CheckStyle-type behavior, and its memory consumption grows cubically with features. This suggests that CtxList replacement would obviate the need for the manual modifications previously necessary to analyze CheckStyle and similar programs [6]. The other benchmarks showed similar gains in performance, universally reducing the naive list's exponential growth to linear or quadratic growth. [2]

6 Related Work

The CtxList implementation was inspired by Walkingshaw et al.'s [11] OList. This work fits into the broader context of variational execution [5, 6, 11] and specifically data structure optimizations for variational execution [7, 11]. Research on variational execution has its roots in model checking [2, 9, 10] and symbolic execution [8]. This work further relates to program optimization in the context of compilers, specifically equivalent rewrites [1].

7 Limitations and Future Work

We implemented automatic substitution of CtxList in a variational execution framework and evaluate its performance gain in micro-benchmarks. The results suggest that the substitution may provide strong improvements for programs processing highly variable lists. However, our benchmarks may not be sufficiently representative to indicate similar performance gains for real systems. Additionally, only iterator loops can currently be transformed. Future work will explore transformation of more types of loops and the performance gains available for real-world systems.

8 Acknowledgments

---

[1]Code available: github.com/chupanw/vbc/tree/iteration-optimization

[2]These results are publicly available at llazarek.github.io/ctxlist-evaluation.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley.

[2] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. 2008. Delta execution for efficient state-space exploration of object-oriented programs. *IEEE Transactions on Software Engineering* 34, 5 (2008), 597–613.

[3] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[4] James A Kupsch and Barton P Miller. 2014. Why do software assurance tools have problems finding bugs like heartbleed? *Continuous Software Assurance Marketplace* 22 (2014).

[5] Jens Meinicke. 2014. *VarexJ: A Variability-Aware Interpreter for Java Applications*. Master's thesis. University of Magdeburg.

[6] Jens Meinicke, Chu-pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity : Measuring Interactions in Highly-Configurable Systems. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 483–494.

[7] Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. 2017. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '17)*. ACM, New York, NY, USA, 28–35. https://doi.org/10.1145/3023956.3023966

[8] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. 2010. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 445–454.

[9] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*. ACM, 842–853.

[10] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. 2009. Efficient online validation with delta execution. *ACM SIGARCH Computer Architecture News* 37, 1 (2009), 193–204.

[11] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. ACM, 213–226.