Addendum to Rewriting for Symbolic Execution of State Machine Models

J Strother Moore*

Department of Computer Sciences, University of Texas at Austin,
Taylor Hall 2.124, Austin, Texas 78712
moore@cs.utexas.edu telephone: 512 471 9568
WWW home page: http://www.cs.utexas.edu/users/moore

Abstract. This note provides some elaborations, optimizations, and proposals for future work related to the algorithm described in our paper "Rewriting for Symbolic Execution of State Machine Models."

Keywords: hardware modeling, verification, microprocessor simulation, theorem proving, pipelined machine

1 Elaborations

An elaboration of the algorithm described in [2] involves how to determine whether two facets represent the same term, as when we compare facets ϕ_1 and ϕ_2 (in paragraph 3.1.1) to determine whether we can use the `(if x y y)` rule, or when we compare facets $\hat{\imath}$ and $\hat{\jmath}$ while applying `nth-update-nth` (in paragraph 3.2). The cheapest test is whether the two facets have identical terms and identical stacks. It is, of course, possible to answer the question directly: literally compare the two terms represented by the two facets, without actually producing the two terms, by exploring the two recursively, chasing variable bindings through the stacks. This test is much more expensive and seldom wins when the cheaper test loses. We use the cheaper test.

The ν-rewriter evaluates ground terms when they arise as facets. For example, the facet $<$`(logand x 7)`$, \rho >$, where ρ binds `x` to `11` becomes the empty facet for `3`.

More interestingly, we allow a large class of unconditional rewrite rules to be applied to the non-`nth` terms visited by the ν-rewriter. That is, before returning the result described here (e.g., the reconciliation of `(nth` $\hat{\imath}$ $\hat{t}$`)` as in paragraph 3.5) the algorithm applies additional rewrite rules.

This requires coding the matching algorithm that checks whether a pattern term, e.g., `(phase x (f x) s)` can be instantiated to match the term represented by a facet, e.g., $<$`(phase a b s)`$, \rho >$. If so, it produces a "substitution"

* This work was supported in part by Advanced Technology Center, Rockwell Collins, Inc., Cedar Rapids, Iowa.

in which variables in the pattern are bound to facets. We then reconcile the right-hand side of the rule under that substitution and rewrite the resulting facet.

Because we want this process to be fast and non-explosive, we do not apply permutative rules (e.g., rules that just permute variables in a term), recursive definitions, conditional rewrite rules, rules that duplicate variables (as would `(f x)` = `(g x x)`), or rules that introduce `if` expressions. ACL2 already identifies this class of rules as *abbreviations*. Using abbreviations in in ν-rewriter affects the use of the cache since the ACL2 user can disable the use of a rule.

2 Integration with ACL2's Rewriter

The algorithm as described above was designed to be used within the standard ACL2 rewriter. The plan called for ACL2's rewriter to call the ν-rewriter on every `nth` expression (before the arguments are rewritten). The idea was that the ν-rewriter will simplify the `nth` expression just one level and return a facet which is `lambda` abstracted back to a term that is then rewritten by the standard rewriter. This allows the mixing of ν-rewriting and standard conditional rewriting, with the latter's consideration of context, type information, linear arithmetic, etc. The tests in the `if` generated in paragraph 3.2.3 are frequently dispatched by the standard rewriter so that only one branch of that `if` is pursued in practice.

However, we have learned that it is cost effective to make a prior recursive pass through the term with the ν-rewriter, rewriting every subterm, before attacking the result with the standard rewriter. This usually eliminates large amounts of irrelevant symbolism.

To code this we added a "recursive mode" to the ν-rewriter that causes it to descend into terms that are not themselves targets of the `nth-update-nth` rule (but which might contain targets). The algorithm admits an interesting optimization when used recursively: all the facets returned have empty stacks. This makes reconciliation unnecessary but risks the production of an exponentially large expansion because all `lambda`s have been eliminated.

Our current integration of the ν-rewriter into the ACL2 system uses the ν-rewriter in recursive mode to simplify each literal of every goal clause. We then apply the standard rewriter to the resulting term. (This is done conditionally on a user supplied flag as mentioned below and provided the literal contains a potential target of the ν-rewriter, e.g., an occurrence of `nth`.) When the standard rewriter then encounters an `nth` term that is in the top-level literal being rewritten, it does not use the ν-rewriter on it, since it has already been so rewritten. But the standard rewriter may encounter other `nth` terms while tentatively expanding recursive functions, backchaining, or after the application of rewrite rules to terms in the literal. Such `nth` terms are rewritten with the ν-rewriter in non-recursive mode, as originally envisioned. This is the default use of the ν-rewriter and is has been found to be quite effective on the industrial scale problems to which it has been applied.

3 Future Work

We are still "tuning" our integration of the algorithm, focusing on tactics for using it and certain low-level implementation details. Of particular interest are the management of the cache and the associated hashing function used to cache Lisp s-expressions. For example, should the ν-rewrite cache be cleared after each subgoal is simplified or should the cache persist for an entire proof? (We currently do the latter but guard the entrance to the cache with a test on the theory under which all the cached ν-rewrites were done.)

The current implementation of caching is done non-applicatively, which is at variance with our preferred programming style within the ACL2 implementation. We hope to eliminate the non-applicative code currently supporting our hash arrays. Rob Sumners has used ACL2's single-threaded objects [1] to implement an efficient, applicative hashing scheme in support of a verified BDD manager [3].

We are investigating better `lambda` abstraction strategies to capture common subexpressions so as to avoid explosions in recursive mode.

Finally, we are exploring the more aggressive use of rewriting on facets and the full generalization of the algorithm for arbitrary rewrite rules.

References

1. R. S. Boyer and J S. Moore. Single-threaded objects in ACL2. *(submitted for publication)*, 1999.
2. J S. Moore. Rewriting for symbolic execution of state machine models. In *Computer-Aided Verification – CAV '01*, volume ???? of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. See URL `http://www.cs.utexas.edu/users/moore/-publications/nu-% -rewriter`.
3. R. Sumners. Correctness proof of a bdd manager in the context of satisfiability checking. In *Proceedings of ACL2 Workshop 2000*. Department of Computer Sciences, Technical Report TR-00-29, November 2000. `http://www.cs.utexas.edu/-users/moore/acl2/workshop-2000/final/sumners2/paper.ps`.